Network of Excellence on
Engineering Secure Future Internet
Software Services and Systems

Network of Excellence

Deliverable D6.4

Security requirement patterns for Future Internet applications, and improved modelling of the scenarios

European Commission
Information Society and Media

Project Number	:	256980
Project Title	:	NESSoS
Deliverable Type	:	Report

Deliverable Number	:	D6.4
Title of Deliverable	:	Security requirement patterns for Future Internet applications, and improved modelling of the scenarios
Nature of Deliverable	:	R
Dissemination Level	:	PU
Internal Version Number	:	0.18
Contractual Delivery Date	:	30th September 2013
Actual Delivery Date	:	31st October 2013
Contributing WPs	:	WP 6
Editor(s)	:	Federica Paci (UNITN), Le Minh Sang Tran (UNITN)
Author(s)	:	Federica Paci (UNITN), Le Minh Sang Tran (UNITN), Kristian Becker (UDE), Francisco Moyano (UMA), Carmen Fernandez Gago (UMA), Riccardo Scandariato (KUL), Koen Yskout (KUL)
Reviewer(s)	:	Jorge Cuellar (SIEMENS), Marianne Busch (LMU), Marinella Petrocchi (CNR)

Abstract

Future Internet (FI) applications are the result of composing services and data from different parties. The development of such kind of applications often involves multiple stakeholders, whose requirements might partially conflict with others'. This raises the need of new techniques to identify all stakeholders and their high-level requirements, as well as potential conflicts. Moreover, since these parties do not necessarily fully trust each other, it is also important to identify potential threats within FI applications.
In this deliverable we present a framework including the modelling and analyses which support the requirements engineers: (1) *in the elicitation of stakeholders and their requirements,* (2) *in the identification of potential conflicts among their requirements, and* (3) *in the identification of potential risks associated with stakeholders' assets.*
The framework employs and extends the Si modelling language and UML profile with trust and reputation constructs. These modelling languages are used as basis to produce a catalogue of patterns that provides support to the security requirements engineers in identifying possible conflicts and threats. We also present a systematic methodology to apply these patterns to elicit and analyse security requirements for FI applications.*
We further address the system sustainability by taking into account the evolutionary aspect of the development process. We use change scenarios to express the co-evolution of requirements and architectures. We model these scenarios by using the extended version of the Si modelling language. By prioritizing change scenario instances, software architects can be able to select a suitable solution (i.e., an implementation choice) that will enable the co-evolution happens while limiting the impact on the system.*

Keyword List

Si language, UML profile, goal-oriented model, security requirement patterns, cloud patterns, requirements conflicts, insider threats, change patterns, solution selection*

Document History

Version	Type of Change	Author(s)
0.1	Deliverable Outline	F.Paci (UNITN)
0.2	Section Modelling Language added	F.Moyano (UMA)
0.3	Section Modelling Language revised	F.Moyano (UMA)
0.4	Section Context and Security Requirement Patterns added	F.Moyano (UMA)
0.5	Section Context and Security Requirement Patterns revised	F.Moyano (UMA)
0.6	Section Context and Security Requirement Patterns revised	K.Beckers (UDE)
0.7	Section From Security Requirements Patterns to Architectural Patterns added	K.Yskout (KUL)
0.8	Section Modelling Language, section From Security Requirements Patterns to Architectural Patterns revised	F.Paci (UNITN)
0.9	Section Introduction, section The eHealth Case Study added	F.Paci (UNITN)
0.10	Section Modelling Language, Context and Security Requirement Patterns revised	K.Beckers (UDE)
0.11	Section Modelling Language, Context and Security Requirement Patterns revised	F.Moyano (UMA)
0.12	Section Context and Security Requirement Patterns, section From Security Requirements Patterns to Architectural Patterns revised	F.Paci (UNITN)
0.13	Section A Pattern-Driven Methodology added	F.Paci (UNITN)
0.14	Section Executive Summary, section Relations to Other Work Packages, section Conclusions, section Publications added	L.M.S.Tran (UNITN)
0.15	Section The eHealth Case Study, Modelling Language revised	K.Beckers (UDE)
0.16	Section Background added, other sections are revised according to internal reviewers	F.Paci (UNITN), L.M.S.Tran (UNITN), F.Moyano (UMA)
0.17	Deliverable is revised according to internal reviewers	L.M.S.Tran (UNITN), K.Beckers (UDE), F.Moyano (UMA)
0.18	Minor revision on typos	L.M.S.Tran (UNITN)

Table of Contents

Acronyms

FI Future Internet

RBAC Role-Based Access Control

SDLC Service Development Life Cycle

UML Unified Modelling Language

P2P Peer-to-Peer

IaaS Infrastructure as a Service

PaaS Platform as a Service

SaaS Software as a Service

List of Figures

List of Tables

Executive Summary

This deliverable describes the results achieved by WP 6 in the third year of the NESSoS project. In this deliverable, we present a framework to support requirements engineers in the modelling and analyses on security requirements. This framework helps to 1) elicit stakeholders and their requirements, 2) identify potential conflicts among requirements, and 3) identify potential risks associated with stakeholders' assets.

The framework provides extensions for modelling languages (e.g., Si*, UML). The extensions are used as bases to produce patterns that help to identify stakeholders and their relations. The framework also includes patterns to identify conflicts between security requirements, and to identify potential conflicts from sharing services, resources, or data with non-fully trusted stakeholders. We further outline a systematic methodology applying these patterns to elicit and analyse security requirements for FI applications. Moreover, we describe an approach to improve the system sustainability by supporting software architects in dealing with co-evolution of requirements and architectures.

Hereafter, we summarize the major results in this deliverable.

Modelling language (Section 3) We report the extensions that we proposed to the modelling languages to represent context and security patterns. First, we extend the Si* modelling language with means for expressing properties of assets and trust. Analysts can specify for each asset a sensitivity level and the security properties that should hold for the asset. The trust is incorporated with a trust level and type of permission granted. These extensions are used to model different patterns to detect potential insider threats. Second, we present a Unified Modelling Language (UML) profile that allows the requirements engineer to specify trust and reputation requirements for an application. We propose an extension to UML in order to help requirements engineers and software designers to have a clearer understanding of the trust and reputation requirements of the system-to-be. The extensions are applied to several diagrams: use case diagrams, class diagrams, and deployment diagrams. Third, we present a meta-model for building context-patterns which include common structures and stakeholders for several different domains. Then we use this meta-model to describe context-patterns.

Trust-aware cloud pattern (Section 4.1) We describe our analysis using trust-aware cloud pattern to evaluate the security of cloud scenarios. We accompany our analysis by templates to systematically gather domain knowledge about the direct and indirect system environments based on the stakeholders' relations to the cloud and to other stakeholders. The outcome of the analysis helps decision makers to decide if a cloud scenario should be pursued or not.

Patterns for requirements conflicts (Section 4.2) We introduce patterns to identify conflicts in requirements models. We consider three types of conflicts: resource conflict, goal conflict, and compliance conflict. A resource conflict is a situation where different instances of a resource are not enough to allow different agents to achieve their goals at the same time. A goal conflict is a situation in which the personal goal of an agent is in contradiction with the goals of the role it plays or when a security goal of an actor collides with the goal of another actor. A compliance conflict arises when the privacy policy of the data collector and/or the data processor does not comply with the consent expressed by a data subject, or to the constraints on the data collection and disclosure imposed by the existing regulations on privacy and data protection.

Patterns for insider threat detection (Section 4.3) We describe patterns to automatically detect potential insider threats in a requirements model. An insider threat can comprise the confidentiality, integrity and availability of an organisation's asset by misusing the permission granted on the asset.

A pattern-driven methodology (Section 5) We present a systematic methodology that applies patterns mentioned in previous sections to elicit and analyse security requirements for FI applications. The methodology includes three phases. First, trust-aware cloud patterns are applied to identify system stakeholders and the trust relationships among them. Then, a requirements model represented as a Si* model is generated. Second, different pattern-based analyses are applied to identify potential problems in the requirements model. Finally, security requirements engineers have to resolve identified problems and update the Si* model.

From security requirements patterns to architectural patterns (Section 6) We describe an approach to support software architects in dealing with co-evolution of requirements and architectures. The approach prioritizes change scenario instances and exploits the asset and trust information in Si* model. This information is used to determine the relevance and importance of a change scenario instance and to select a suitable solution for the scenario. We illustrate the approach using the eHealth case study from WP11.

1 Introduction

FI applications are the result of composing services and data from different parties. Such applications often involve multiple stakeholders and each one has his own requirements. However, different stakeholders might have conflicts between their requirements. Moreover, since these parties do not necessarily fully trust each other, it is also important to identify potential threats within the FI applications.

Therefore, it raises the need for new techniques to:

- elicit all stakeholders and their high-level requirements,
- identify conflicts among their security requirements,
- early identify potential threats arising from sharing services, or resources, or data between stakeholders who are not fully trusted.

This deliverable reports the results of WP 6 during the third year within the NESSoS project. The main artefact of WP6 is a framework to support requirements engineers in the modelling and analyses on security requirements. Particularly, these results are as follows:

- The extensions of modelling languages (e.g., Si*, UML) in order to elicit domain knowledge (e.g., stakeholders), and capture security requirements at different levels of abstraction,
- Patterns to elicit stakeholders and their relations,
- Patterns to identify conflicts in security requirements,
- Patterns to early identify potential threats associated with stakeholders' assets,
- A pattern-driven methodology to apply these patterns to elicit and analyse security requirements for FI applications,
- An approach to improve the system sustainability by supporting software architects to deal with co-evolution of requirements and architectures.

This deliverable is organised as follows. Section 2 discusses the case study which is used for illustration purpose and briefs the background knowledge for consecutive sections. Section 3 describes the extensions in Si* and UML, and a meta-model for context-patterns. Section 4 presents patterns to identify requirements conflicts, and patterns to identify potential threats. Section 5 outlines the pattern-driven methodology. Section 6 describes an approach to support software architects in the prioritization of instances of change patterns and in the selection of suitable architectural solutions. Section 7 shows the relation between this deliverable and other WPs. Section 8 concludes the deliverable. Section 9 lists the publications of WP 6 in the third year of the NESSoS project.

2 Background

This section, as name suggested, provides background knowledge for further sections in this deliverable. Section 2.1 describes the eHealth case study which is used for the illustration purpose in later sections. Section 2.2 presents the basic principle and concept of the Si* language. Section 2.3 briefs studies in the field of trust and reputation requirements. Section 2.4 describes several patterns that we have developed for the elicitation of domain knowledge. Section 2.5 discusses the usage of change pattern for the co-evolution of requirements and architectures of a system.

2.1 The eHealth Case Study

To illustrate the patterns that will be described in later sections, we use the health care record management and the patient monitoring scenarios. We briefly recall each scenario in what follows.

Patient Monitoring The scenario concerns health monitoring and drugs delivery to patients' home. The scenario involves six actors. The **Patient** is monitored by a smart T-shirt which measures medical data (e.g., heartbeat rate, blood pressure, etc.) and transfers them to the Hospital's computer system. When the patient's condition is abnormal, the doctor makes a diagnosis and produces a prescription. The patient receives his prescription and requests the drug delivery service to the pharmacy. The **Hospital** provides medical services to patients. The hospital monitors patients' health and manages patients' data, which are stored in the hospital's computer. When the patient has some problems, the hospital assigns a doctor to diagnose the patient. The **Doctor** is responsible to diagnose the patients and prescribe required medications. The **Pharmacy** sells drug to patients; it is responsible for managing drugs and provide them to patients. All the information about drugs is stored in the pharmacy's computer. The **Pharmacist** works for the pharmacy and communicates directly to patients to receive prescription from them. The pharmacist is responsible to provide drugs to be delivered according to patients' prescriptions. The prescription information is stored in the pharmacy's computer. The **Drug manager** works for the pharmacy and is responsible to manage the pharmacy's computer which stores all the drugs' information. The drug manager does not directly communicate to patients. If a patient has any concern to drug delivered for him by the pharmacy, he can contact the pharmacist instead of the drug manager.

Electronic Health Records (EHR) The scenario concerns managing *Electronic Health Records* (EHRs) and is provided by the industrial partners of the EU project NESSoS. EHRs contain any information created by health care professionals in the context of the care of a patient. Examples are laboratory reports, X-ray images, and data from monitoring equipment. The information stored in the EHR shall only be accessed with the consent of the patient. The only exception is a medical emergency, in which case the patient's physical status may prevent her from giving the consent. In addition, the information in the EHR can support clinical research.

2.2 The Si* Language

The Si* modelling language [47] has been proposed to capture security and functional requirements of socio-technical systems. We do not introduce here all the concepts and relations of Si*, but just the ones that are relevant to illustrate our approach. We consider the concepts *agent*, *role*, *service*, and the relations *AND/OR decomposition*, *means-end*, *contribution*, and *delegation* and *trust* of execution and permission.

An *agent* is an active entity with concrete manifestations and is used to model humans as well as software agents and organisations. A *role* is the abstract characterization of the behaviour of an active entity within some context. They are graphically represented as circles.

The term *service* is used to denote a *goal*, a *task* and a *resource*. A *goal* captures a strategic interest that is intended to be fulfilled. A *task* represents a particular course of actions that produces a desired effect. It can be executed to satisfy a goal. A *resource* is an artefact produced/consumed by a goal or a

task. In the graphical representation, goals, tasks, and resources are respectively represented as ovals, hexagons, and rectangles.

AND/OR decomposition is used to refine a goal, while *means-end* identifies goals that provide means for achieving another goal or resources produced or consumed by a goal/task. *Contribution* is used when the relation between goals is not the consequence of a deliberative planning but rather results from side-effects. The impact can be positive or negative and is graphically represented as edges labelled with + and -, respectively.

The *delegation* and *trust* are used to capture the relation between actors. A *delegation* relation between two actors marks a formal passage of responsibility (*delegation execution*) or authority (*delegation permission*) from an actor (*delegator*) to another actor (*delegatee*) who receives the responsibility/authority to achieve a goal or to provide a resource. Usually, an actor prefers to appoint actors that are expected to achieve assigned duties and not misuse granted permissions. Si* adopts the notions of *trust of execution* and *trust of permission* to model such expectations. *Trust of execution* is a relation between two actors representing the expectation of one actor (*trustor*) about the capabilities of another actor (*trustee*), while *trust of permission* represents the belief of the trustor that the trustee will not misuse a given permission. In the graphical representation, delegation execution and delegation permission relations are represented by edges labelled `De` and `Dp` respectively. Similarly, trust of execution and trust of permission relations are represented by edges labelled `Te` and `Tp` respectively.

2.3 Trust and Reputation Requirements Specification

There are several works that consider security requirements at the early stages of the Service Development Life Cycle (SDLC). Some of these works focus on detecting possible attacks on the system [40, 38]. In others, the emphasis is on modelling security requirements, such as confidentiality or authorization. This is the case of UMLsec [23] and SecureUML [25], two UML profiles that include security constraints and annotations into the diagrams. Other works aim to integrate the notion of risk into the requirement analysis stage [27] in order to assess whether the risk level of some unwanted incidents is beyond an acceptable threshold.

The contributions mentioned up to now focus on hard security requirements or risk, but they usually lay trust aside. In addition to traditional policy languages for distributed trust management [13, 21], there are other works that focus on trust in early stages of the SDLC. Mouratidis and Giorgini [29] present Secure Tropos, a methodology that extends the Tropos methodology in order to enable the design of secure systems. Actors in Tropos may depend on other actors in order to achieve a goal, and these social relationships are captured by the methodology. In a similar direction, Lamsweerde and Letier present KAOS [44], a comprehensive goal-oriented methodology to elicit the requirements of a socio-technical system. All these contributions put forward the idea of capturing social aspects, but the notion of trust and its influence on the information systems is barely explored. This is partially covered by Pavlidis, Mouratidis and Islam [36], who include trust-related concepts in Secure Tropos.

The work by Chakraborty and Ray [14] bridges a gap between traditional security requirements modelling and soft-security considerations by incorporating the notion of trust levels into the traditional Role-Based Access Control (RBAC) model. These levels are measured by means of a trust vector, where each component in the vector is a factor that influences trust, such as knowledge or experience.

In general, the aforementioned works usually fail in capturing and making explicit all the trust relationships, and above all, how trust and reputation can be used by the system-to-be. The closest contribution to our work is the one by Uddin and Zulkernine [43], who present a UML profile for trust called UMLtrust. They provide extensions, as we do, to some UML diagrams in order to represent trust information. Their approach and focus is, however, different than ours. First, their primary concern is reasoning about *trust scenarios*, without making explicit which are the trust relationships in the system. Also, they do not address reputation, whereas it is a primary concern for us. We also provide more details on how trust and reputation can be computed and the factors (e.g., variables and attributes) that will be taken into account for this computation. We also show how trust can influence at the infrastructure level by means of deployment diagrams. However, our trust analysis is in general at a higher level of abstraction, without delving into the details of class attributes and methods, which is something that UMLtrust requires. As a conclusion, we think that both works are complementary and can help each other in providing a more

comprehensive vision of trust in the system for designers and developers.

2.4 Context-Patterns

In the past we developed a number of patterns for the elicitation of domain knowledge, so-called *Context-Patterns*. We describe these patterns, because we show a meta-model for context-patterns in Section 3.3. The meta-model is based on the analysis of our previous context-patterns, which are presented in the following. The meta-model supports software engineers to describe further context-patterns.

2.4.1 P2P pattern

Our Peer-to-Peer (P2P) pattern (see Figure 2.1 top) is based upon the P2P architecture from Lua et al. [26], which is derived from a survey of existing P2P systems. This survey describes P2P systems as layered architectures that contain at least the following layers.

The *Application Layer* concerns applications that are implemented using the underlying P2P overlay. For example, a Voice-over-IP (VoIP) application. The *Service Layer* adds application specific functionality to the P2P infrastructure. For example, for parallel and computing-intensive tasks or for content and file management. Meta-data describe what the service offers, for instance, content storage using P2P technology. Service messaging describes the way services communicate. The *Feature Management Layer* contains elements that deal with security, reliability and fault resiliency, as well as performance and resource management of a P2P system. All these aspects are important for maintaining the robustness of a P2P system. The *Overlay Management Layer* is concerned with peer and resource discovery and routing algorithms. The *Network Layer* describes the ability of the peers to connect in an ad hoc manner over the internet or small wireless or sensor-based networks.

2.4.2 Cloud Pattern

We also briefly introduce our cloud pattern (see Figure 2.1 bottom) [11]. A *Cloud* is embedded into an environment consisting of two parts, namely the *Direct System Environment* and the *Indirect System Environment*. The *Direct System Environment* contains stakeholders and other systems that directly interact with the *Cloud*, i.e., they are connected to the cloud by associations. Moreover, associations between stakeholders in the *Direct* and *Indirect System Environment* exist, but not between stakeholders in the *Indirect System Environment* and the *Cloud*. The *Cloud Provider* owns a *Pool* consisting of *Resources*, which are divided into *Hardware* and *Software* resources. The provider offers its resources as *Services*, i.e., Infrastructure as a Service (IaaS), Platform as a Service (PaaS), or Software as a Service (SaaS). The *Pool* and *Service* do not require instantiation. Instead, the specialized cloud services such as IaaS, PaaS, and SaaS and specialized *Resources* are instantiated. The *Cloud Developer* represents a software developer assigned by the *Cloud Customer*. The developer prepares and maintains an IaaS or PaaS offer. The IaaS offer is a virtualized hardware, in some cases it is equipped with a basic operating system. The *Cloud Developer* deploys a set of software named *Cloud Software Stack* (e.g., web servers, applications, databases) into the IaaS in order to offer the functionality required to build a PaaS. In our pattern PaaS consists of an IaaS, a *Cloud Software Stack* and a *cloud programming interface (CPI)*, which we subsume as *Software Product*. The *Cloud Customer* hires a *Cloud Developer* to prepare and create SaaS offers based on the CPI, finally used by the *End Customers*. SaaS processes and stores *Data* input and output from the *End Customers*. The *Cloud Provider*, *Cloud Customer*, *Cloud Developer*, and *End Customer* are part of the *Direct System Environment*. Hence, we categorize them as *direct stakeholders*. The *Legislator* and the *Domain* (and possibly other stakeholders) are part of the *Indirect System Environment*. Therefore, we categorize them as *indirect stakeholders*.

2.4.3 The SOA Pattern

Our SOA patterns concerns domain knowledge for Service-Oriented Architectures (SOA). The following description is taken from [9]. A SOA spans different layers [9], which form a pattern on a SOA with technological focus, as depicted in Figure 2.2 on the top. The first and top layer is the *Business Domain* layer,

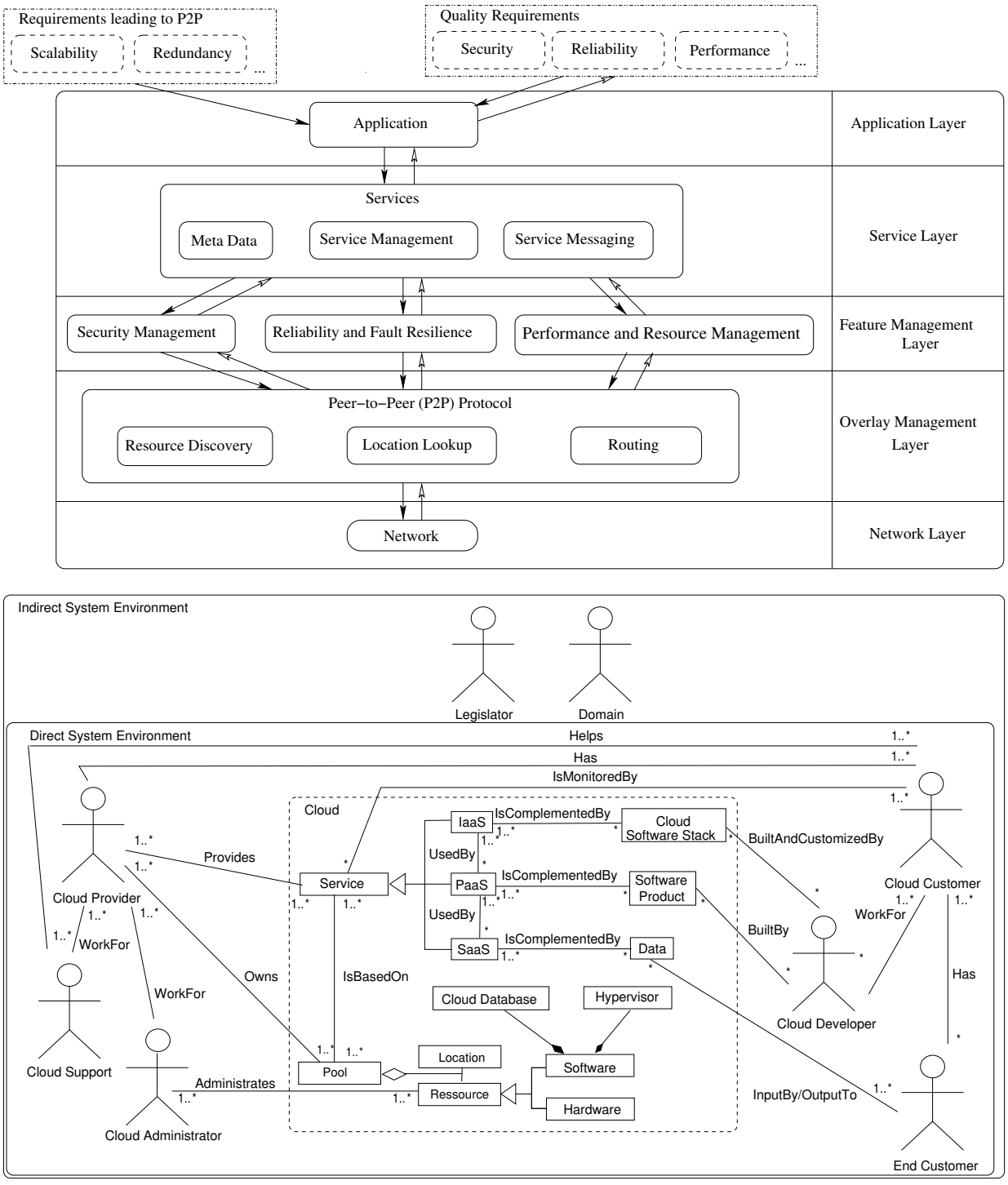

Figure 2.1: P2P pattern (top) and Cloud Analysis Pattern (bottom)

which represents the real world. It consists of *Organisations*, their structure and actors, and their *business relations* to each other. The second layer is the *Business Process* layer. To run the business, certain *Processes* are executed. Organisations *participate in* these processes. These processes are supported by *Business Services*, which form the *Business Service* layer. A business service encapsulates a business function, which *performs* a process activity within a business process. All business services rely upon *Infrastructure Services*, which form the fourth layer. The infrastructure services offer the technical functions needed for the business services. These technical functions are either implemented especially for the SOA, or they expose interfaces from the *Operational Systems* used in an organisation. These operational systems, like databases or legacy systems, are part of the last SOA layer at the bottom of the SOA stack. These layers form a generic pattern, the SOA layer pattern, to describe the essence of a SOA.

In Figure 2.2 on the bottom, we adapted problem-based methods, such as problem frames by Jackson [22], to enrich the SOA layer pattern with its environmental context. The white area in Figure 2.2 (bottom) spans the SOA layers that form the machine. The business processes describe the behaviour of the machine. The business services, infrastructure services, components, and operational systems describe the structure of the machine. Note that the business processes are not part of the machine altogether, as the processes also include actors, which are not part of the machine. Thus, the processes are the bridge between the SOA machine and its environment. The environment is depicted by the grey parts of Figure 2.2 (bottom). The light grey part spans the *Direct Environment* and includes all entities, which participate in the business processes or provide a part, like a component, of the machine. An *entity* is something that exits in the environment independently of the machine or other entities. The dark grey part in Figure 2.2 (bottom) spans the indirect environment. It comprises all entities not related to the machine but to the direct environment. The Business Domain layer is one bridge between the direct and indirect environment. Some entities of the Direct Environment are part of organisations. Some entities of the Indirect Environment influence one or more organisations. The machine and the Direct Environment form the *inner system*, while the *outer system* also includes the Indirect Environment.

The entities we focused on for the stakeholder SOA pattern are stakeholders, because all requirements to be elicited stem from them. There are two general kinds of stakeholders. The *direct stakeholders* are part of the direct environment, while the *indirect stakeholders* are part of the indirect environment. We derived more specific stakeholders from the direct and indirect stakeholders, because these two classes are very generic. Process actors and different kinds of providers are part of the direct environment. Legislators, domains, shareholders and asset providers are part of the indirect environment. In Figure 2.2(bottom), the resulting stakeholder classes are depicted as stick figures. For a detailed description of these stakeholders we refer to our previous work [9].

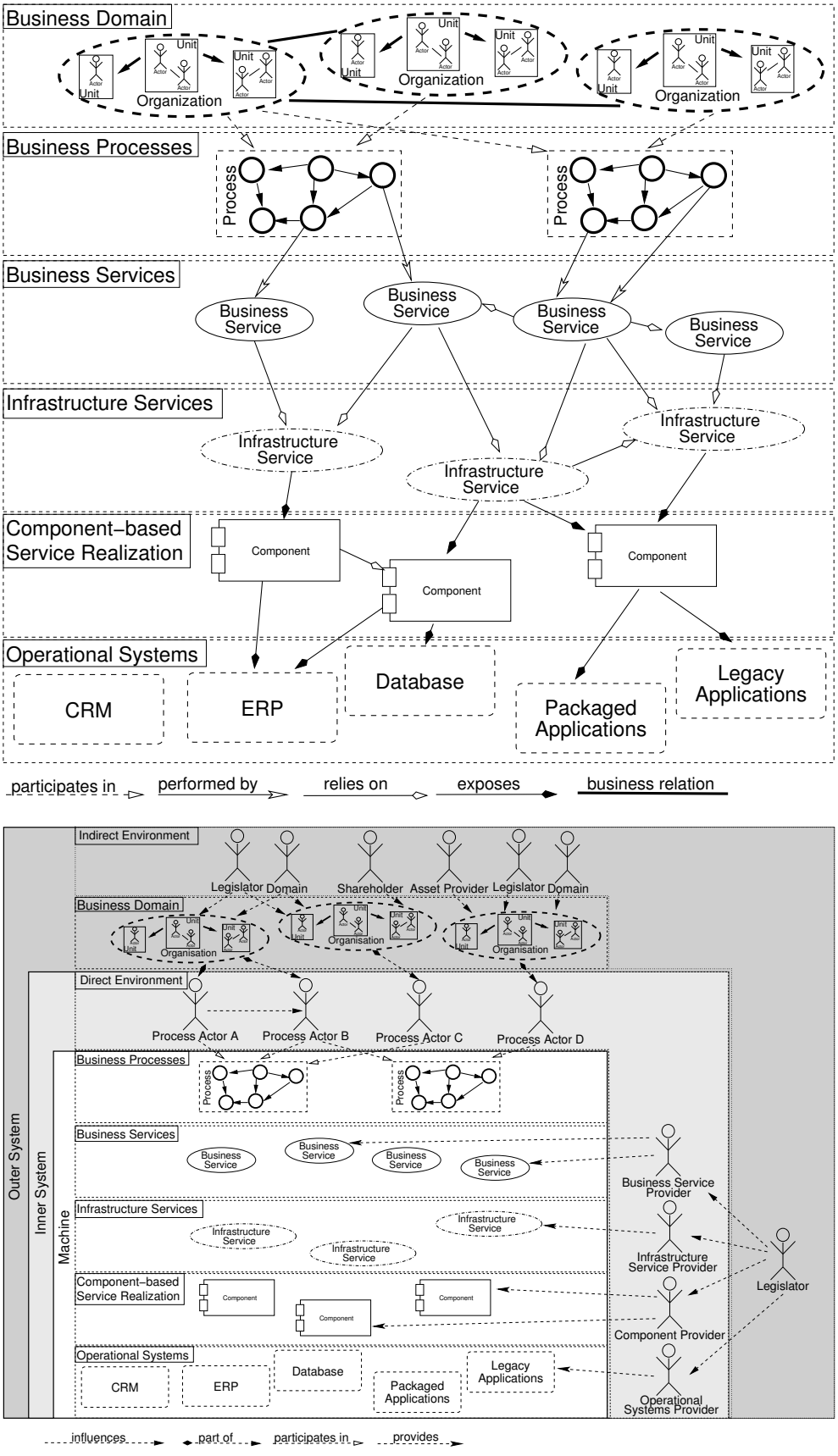

Figure 2.2: SOA Layer Pattern (top) and SOA Layer Stakeholder Pattern (bottom)

2.4.4 Patterns for Requirement-Based Law Identification

We consider legal domain knowledge using a set of law patterns. Commonly, laws are not adequately considered during requirements engineering [34]. Therefore, they are not covered in the subsequent system development phases. One fundamental reason for this is that involved engineers are typically not cross-disciplinary experts in law and software and systems engineering. To bridge this gap we developed law patterns and a general process for law identification which relies on these patterns [10].

We investigated how judges and lawyers are supposed to analyse a law, based upon legal literature research. These insights lead to a basic structure of laws and the contained sections and how they relate to the context of a system-to-be in terms of requirements. One result of our investigations is a common structure of laws. Based on this structure of laws, we defined a *law pattern* (see Figure 2.3 left hand). The law pattern itself is discussed in detail in [10]. Every dictate of justice as part of a law states that every *Addressee*, who avoids or accomplishes a certain *Activity* which influences a *Target Subject* or a *Target Person*), to which an *Individual* (*Target Person*) is entitled to, has to comply with law. This information forms the *Law Structure*. The artefacts of the *Law Structure* are generalized in the *Classification* part. The addressees of this section are specializations of *Person Classifier*, the activities of *Activity Classifier*, the target subject of *Subject Classifier*, and the target person of *Person Classifier*. A Law itself is enacted by a *legislator* for a *Domain* and related to other laws (*Law Pattern*).

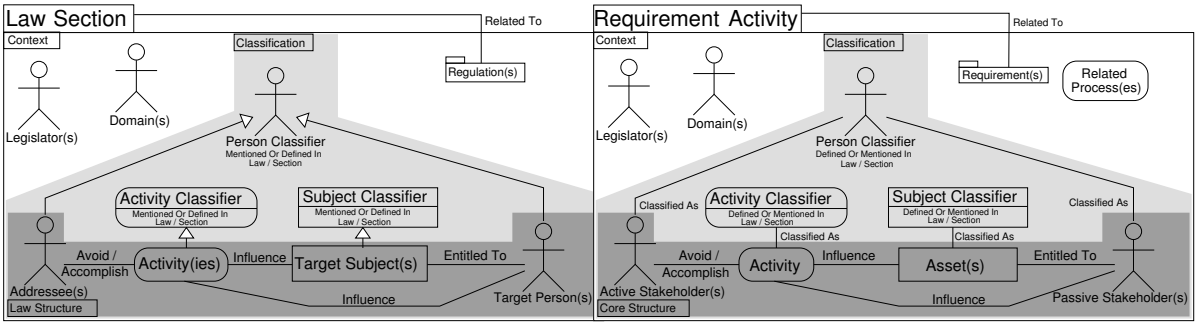

Figure 2.3: Law Pattern (left) and Law Identification Pattern (right)

To find to the legal context of a system the requirements have to be related to the law pattern. Therefore, we developed the law identification pattern (see Figure 2.3 right hand). First of all, a *Requirement* can be related to other *Requirements* and dictates a certain behaviour of the machine. A behaviour can be a certain *Activity* or a whole *Process*. A *Process* consists of different *Activities*. An *Activity* involves an *Active Stakeholder* and in some cases an *Asset*. Additionally, an *Activity* influences a *Passive Stakeholder* in a direct way or indirect through an *Assets* which is entitled to the *Passive Stakeholder*. In addition *Assets* can be related to each other, e.g. one *Asset* is part of another *Asset*. All these relations have also to be discovered and documented. They form the *Core Structure* of the law identification pattern. Then the gap between the terms and notions of the technical world and the terms and notions of the legal world has to be bridged. Therefore, the parts of core structure have to be classified using the terms of the legal world. And the context in means of *Countries* and *Domains* the system will operate in has to be set up.

2.5 Change Pattern

A change pattern [46] represents a reusable source of knowledge concerning the co-evolution of two related artefacts. Depending on the type of artefacts involved, there can be many families of change patterns. For instance, patterns of co-evolution are likely to be found among design and implementation, requirements and test cases, and so on. We focus on change patterns for co-evolving the (security) requirements and the software architecture of a system.

When the requirements of a system evolve, the system's architecture most likely needs to be updated to accommodate the changed requirements. Keeping the requirements and architecture synchronised, such that the system at all times fulfils its requirements, is a big challenge for a software architect. Change patterns can help to achieve this challenge, by providing advice to the architect in the form of a reusable solution.

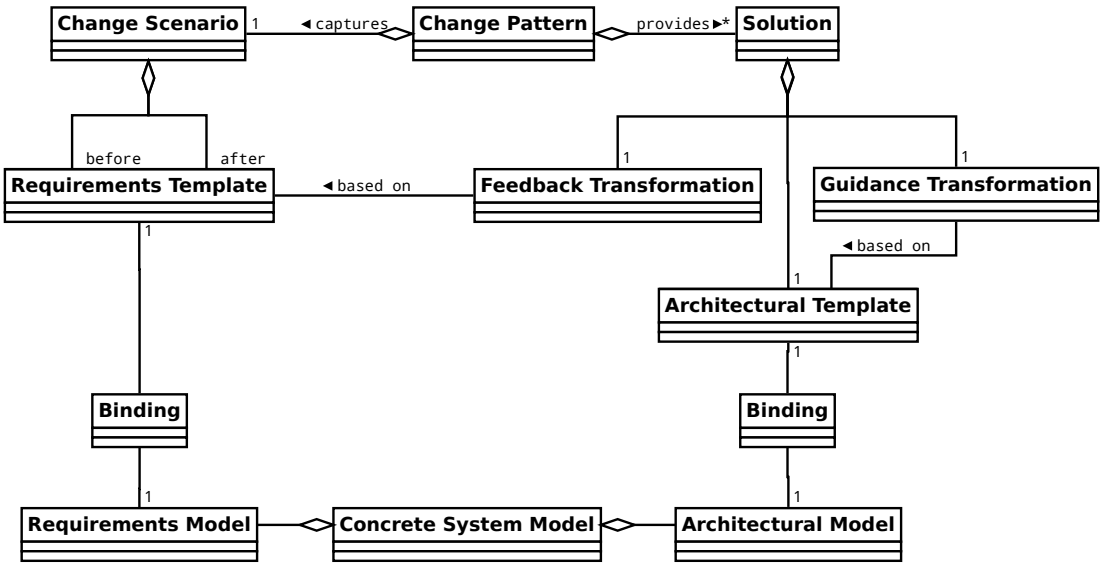

Figure 2.4: Conceptual model of change patterns (expressed in UML)

Figure 2.4 contains the main constituents of a change pattern, and the relationships among them. A possible change at the requirements level is captured by means of a *change scenario*, which consists of a pair of *requirements templates* that describe, in a generic way, the situations before and after the anticipated change. To interpret a scenario in the context of a concrete system, a *binding* needs to be defined in order to link a requirements template to that system's requirements model. These bindings may be defined manually, or they can be found automatically by using pattern matching (such as EMF IncQuery [12]).

Moreover, the pattern provides a collection of *architecture-level solutions*. Each solution provides an alternative way for the system to respond to the change, while minimizing the required effort to evolve the architecture. Additionally, the principled solutions suggested by a change pattern aim at reducing the impact (in terms of disruptive change) of the evolution. This is important when the system that evolves has already been deployed, and recalling the system to carry on major changes to the architecture is prohibitive. Hence, the use of change patterns helps a software architect in creating a sustainable architecture.

The solution is composed of a generic *architectural template* and a transformation called the *guidance*. The guidance is specified in terms of the elements of the architectural template, which serves as a reference point to apply the guidance. A binding between the architectural template and a concrete system's architectural model is used to apply the guidance to the concrete system's architecture. Changing the architecture (by applying the guidance) may, in turn, necessitate modifications to the requirements model. To express this, the solution also contains a *feedback* transformation, which captures the influence of the architectural changes on the requirements model. Similar to the guidance, the feedback transformation is specified in terms of a (requirements) template that needs to be linked to the concrete requirements model by means of a binding.

Figure 2.5 reports a summary of the change patterns that have already been defined for evolving trust-of-permission relations using Si* as modelling language for the change scenarios [45].

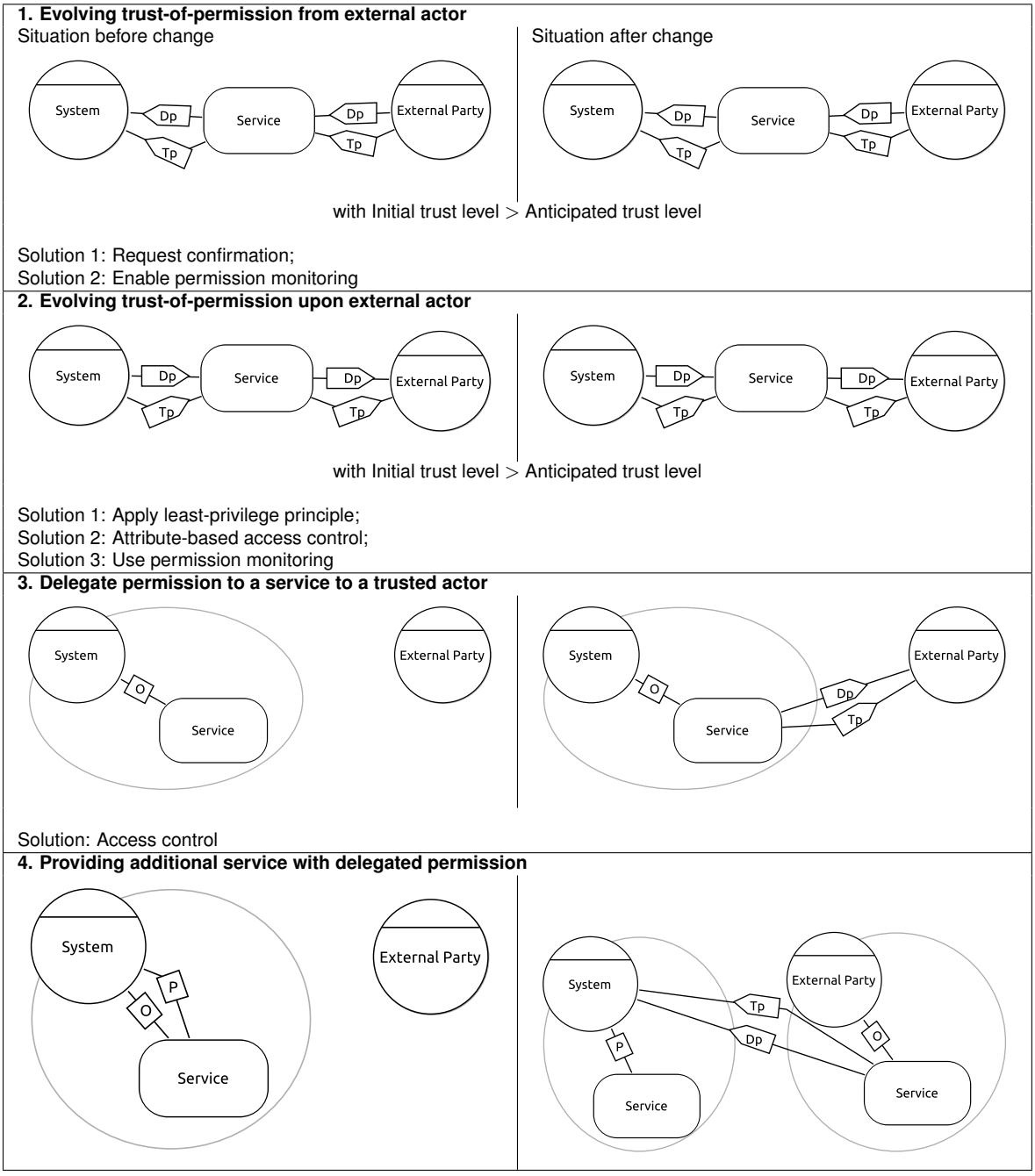

Figure 2.5: Overview of change patterns for evolving trust-of-permission (adapted from [45])

3 Modelling Language

In this section we discuss our extensions in modelling languages to represent domain knowledge, requirements conflict patterns and patterns to detect insider threats. Section 3.1 describes extensions to the Si* language which patterns in later sections are built upon. Section 3.2 presents an UML profile that allows the requirements engineer to specify trust and reputation requirements for an application. This involves making trust relationships and reputation information explicit, including the ways they are to be updated and the interrelations to the business logic of the application. Section 3.3 presents our meta-model for context-patterns. The meta-model shows a common structure of our context-patterns, which are introduced in Section 2.4. These patterns support the structured consideration of domain knowledge during requirements engineering. Software and security engineers can use our meta-model to describe their own context-patterns. Thus, we contribute a basis for software and security engineers to share their domain knowledge with other software engineers and security experts.

3.1 The extended Si* Language

The Si* modelling language [47] can only capture whether an actor has a permission (either delegation permission, or trust permission) on a resource or not but it does not allow one to specify which is the type of permission the actor is granted on the resource (see Section 2.2). In [5], Si* has been extended with *permission type* to represent different types of actors' permissions on resources. Specifying the permission type is crucial because it determines the type of actions an actor can perform on a resource. Some of these actions might be used by an actor to un/intentionally harm a resource. Therefore, we refine the notion of permission in Si* by introducing three different types of permissions: *access*, *modify*, and *manage* as described in Table 3.1.

Table 3.1: Permissions on resource

Permission Type	Description	(Possible) Affected Security Property
Access (low-level)	Actor only has the permission to access/read/use the resource.	Confidentiality
Modify (medium-level)	Actor can change the content of the resource.	Integrity
Manage (high-level)	Actor has the permission to modify the resource, delegate permissions to other actors and modify permissions to other actors.	Availability

Each *permission type* determines the set of actions that an actor can perform on a resource. Thus, a permission type might lead to the violation of a specific security property if the actor misuses the actions associated with the permission type. In Table 3.1 we show the relation between permission types and security properties that might be violated if an actor abuses of the permission type. For example, if an actor has an access permission on a resource, he can accidentally disclose the resource and thus violate the resource's confidentiality. In the graphical representation, permission types *access*, *modify*, and *manage* are respectively represented by suffixes `_a`, `_m`, and `_ma`. These suffixes concatenate the label `Dp` or `Tp` on edges that graphically represent the delegation/ trust permission relations.

In [35], we proposed a further extension to Si* with means for expressing properties of assets and trust. An asset can be a service (e.g., a goal, task, resource). The owner of an asset specifies a *sensitivity level* (VERY LOW, LOW, MEDIUM, HIGH, VERY HIGH) and a *security property* (e.g., confidentiality, availability, integrity) that expresses the need of protecting the asset. The trust permission relationship is incorporated with a *trust level* (e.g., VERY GOOD, GOOD, NEUTRAL, BAD, or VERY BAD) and the *permission type* granted (e.g., *access*, *modify*, *manage*).

The sensitivity level and the trust level with which an agent is granted a given permission on an asset are used to quantify the risk associated with the agent misusing the granted permission to cause harm

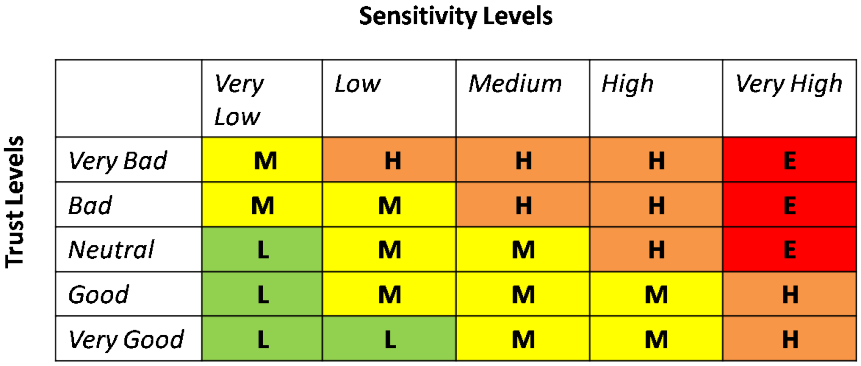

Figure 3.1: Risk Levels

to the asset. The sensitivity quantifies the cost of the threat, while the trust level quantifies the likelihood that the threat occurs. Intuitively, the higher the sensitivity of the asset, the higher the damage for the organisation. Similarly, the higher the trust level, the lower the likelihood that the agent will misuse the granted permission [15]. The risk level can assume the values LOW, MODERATE, HIGH, or EXTREME, based on the values assumed by sensitivity and trust levels as illustrated in Figure 3.1.

3.2 From Trust and Reputation Patterns to a UML Profile for Trust and Reputation

FI scenarios usually comprise a huge number of heterogeneous, geographically distributed entities, including human users, which must interact to provide services. The complexity of managing security in these scenarios is aggravated by their dynamic nature, with devices changing, appearing and disappearing along the system lifetime. In these complex and open scenarios, new security requirements need to be tackled beyond the traditional hard security requirements: confidentiality, integrity and availability.

Trust management is a soft security mechanism [37] that can leverage the security of a system. Even though there is not any accepted definition of trust, it is agreed that it can improve decision-making processes under risk and uncertainty, improving in turn systems security. Reputation, which is a concept strongly related to trust, can also help in this task. We argue that increasing security in FI applications entails that trust relationships between actors, applications and system environments cannot be taken for granted anymore and must be explicitly specified from the very beginning in the SDLC. However, security requirements engineering methods often do not consider trust requirements, but focus on hard security ones such as confidentiality or authorization [23, 25]. Even when some social aspects are beginning to be captured at the requirements stage [29, 44], the analysis of trust relationships is still naive. This could explain why trust and reputation models have been traditionally added after-the-fact in an ad-hoc fashion, limiting their re-usability and presenting scalability problems [18].

We advocate that a comprehensive analysis of trust and reputation during the initial stages of the SDLC is required. For this reason, we propose an extension to UML in order to help requirements engineers and software designers to have a clearer understanding of the trust and reputation requirements of the system-to-be.

The performed extensions are the result of identifying trust and reputation patterns by means of a conceptual analysis [30]. We elicit elements and concepts that are part of most trust and reputation models and find their relationships. This domain knowledge facilitates the shift from a conceptual domain to a technical, UML domain.

The extensions are applied to several diagrams:

- Use case diagrams: the goal of the extensions is to depict, at a glimpse, the trust relationships that exist between the different actors in the system. We also use this diagram to depict which actors can make claims about which other actors, thus incorporating reputation information.

- Class diagrams: the extensions to these diagrams provide more insight in trust relationships, including how they are represented and how they are updated. They also give more details on claims and the way to compute them to yield a reputation value about a target entity.

- Deployment diagrams: besides representing the software from the infrastructure point of view, the extensions to these diagrams show the trust relationships between these infrastructure elements and their reputation information. Platforms and networks can trust each other and they can even hold reputation values. This is particularly useful when designing large-scale distributed systems, where a given processing node (e.g., a mobile phone or a server) can choose among different nodes in order to collaborate.

We also consider indispensable the use of some behavioural diagram, such as activity diagrams, even though we do not propose extensions for them. This diagram should represent the interaction patterns between the trust and business logic of the application, and should make clear which actor initiates a trust event, how this event is triggered and its consequences. For us, a trust event is any occurrence in the system that triggers a reputation or trust relationship update.

In order to consider trust and reputation requirements early in the SDLC, we will present next how we can apply the UML profile to the eHealth case study that was introduced in D11.3 [2]. This case study aims to collect health-related data independently of the location of the *Patient*. This is useful for *Patient*(s), who can receive immediate feedback under critical situations and be assisted by *Physician*(s) at any moment and place. In order to make this scenario feasible, the *Patient* must wear a device capable of measuring vital signs (e.g., blood pressure). This device must be able to send this information to other systems that will show it to *Physician*(s) for monitoring purposes if there is any problem. For further information about the actions that can be performed by the *Physician* and by the *Patient*, and the original use case diagram, we refer the reader to D11.3 [2].

We focus now on a possible trust-aware use case diagram, as shown in Figure 3.2. We state that there is a trust relationship between the *Patient* and the *Physician*. The *Patient* plays a *trustor* role and the *Physician* plays a *trustee* role. In addition, there is a *trusts* connector, which is adorned by the *context* where this trust relationship is set, namely *monitoring*. There is another trust relationship between the *Physician* (who therefore also plays a *trustor* role) and the wearable. The *Patient* also plays the *source* role and can therefore make claims (*claims* connector) about the *Physician*, who plays in this case the *target* role. Please note that we have also added some relevant trust-related use cases. See [31] for more information on this.

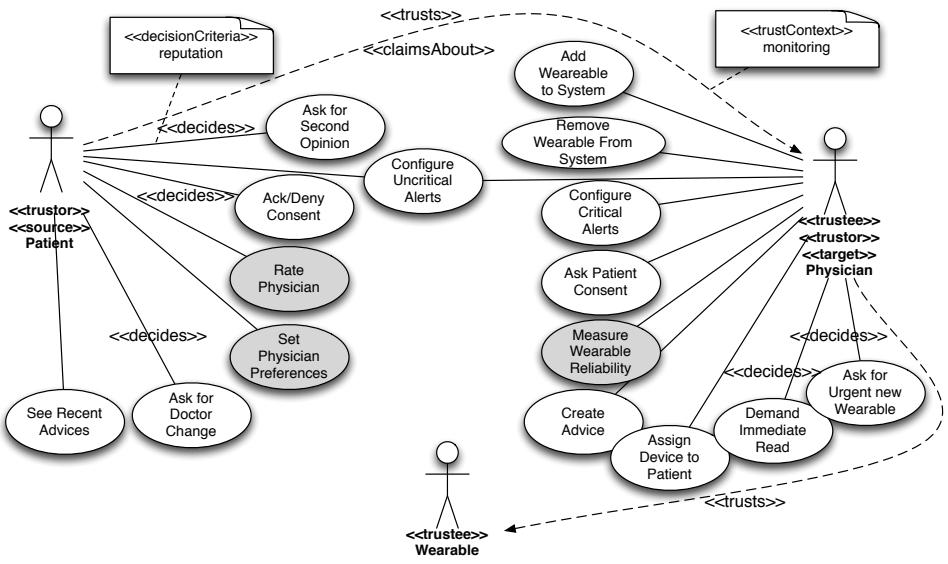

Figure 3.2: Trust-aware use case diagram

Trust relationships and claims can be further refined in trust-aware class diagrams, as shown in Figure 3.3 and Figure 3.4 respectively.

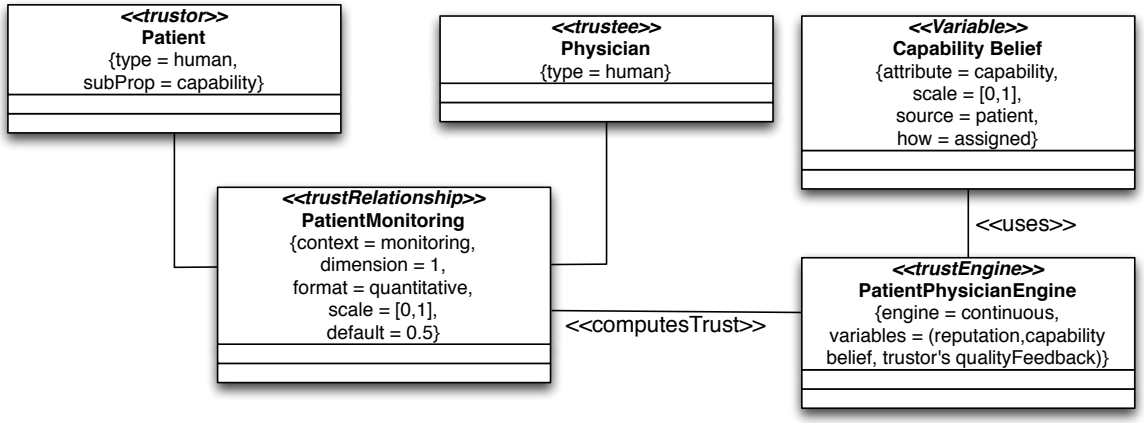

Figure 3.3: Patient-physician relationship

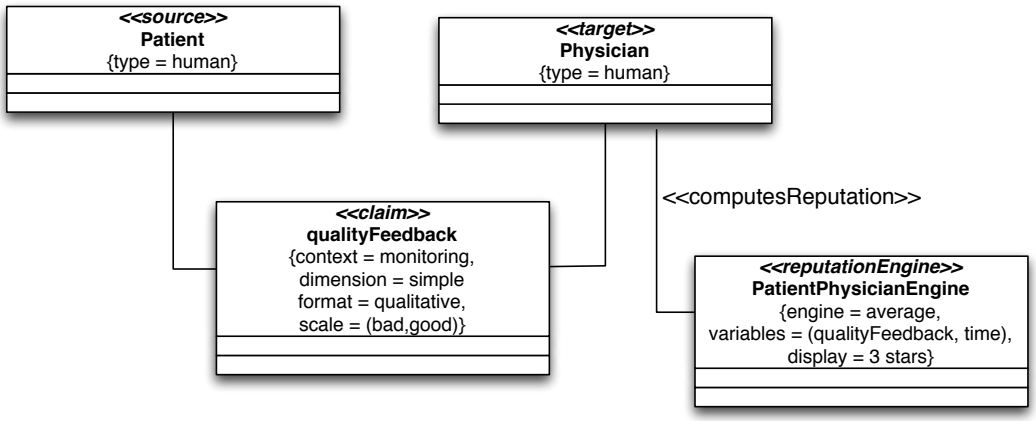

Figure 3.4: Quality feedback claim

In order to specify how the business and trust layers interact, we can define activity diagrams for some use cases. Figure 3.5 shows an activity diagram for the use case *ask for second opinion*.

Finally we specify trust information at the infrastructure level. We basically define on which node the reputation information is to be stored, and how nodes can decide whether to interact with different nodes according to their reputation.

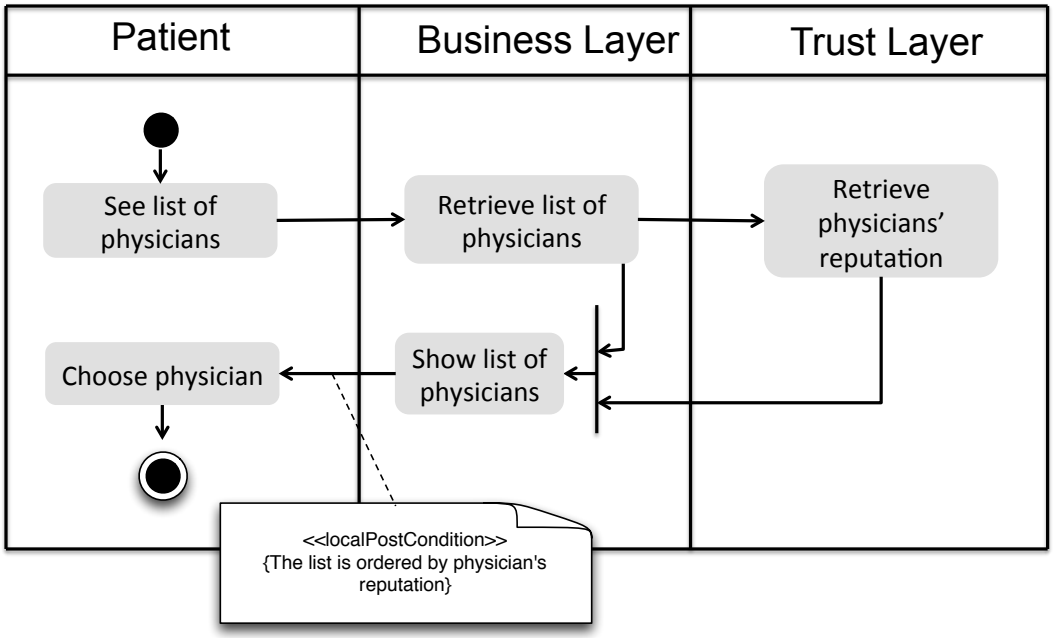

Figure 3.5: Activity diagram for use case *ask for second opinion*

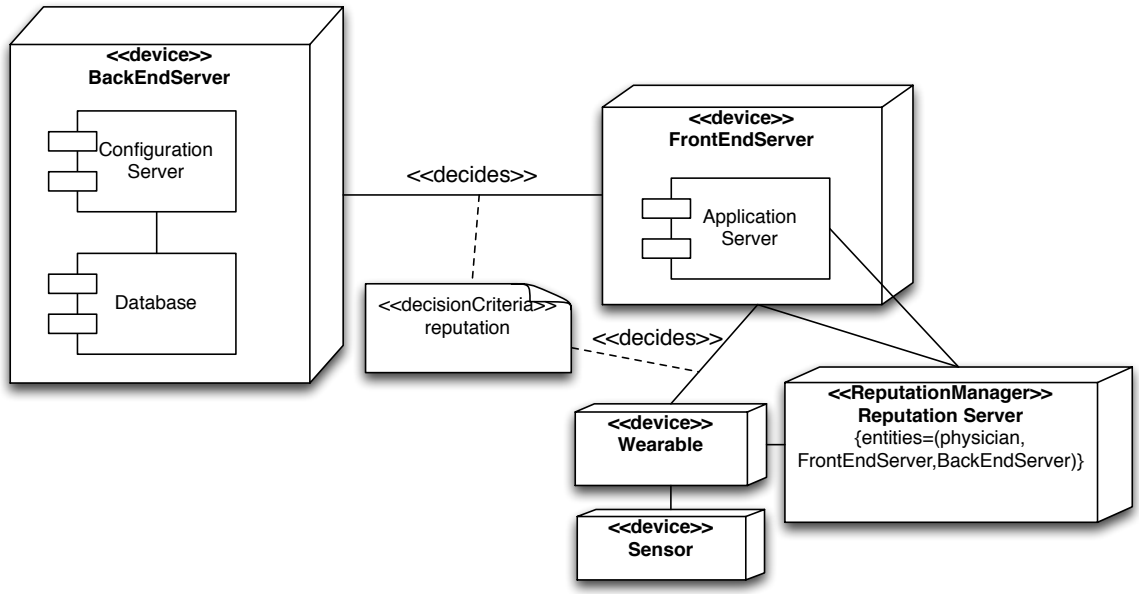

Figure 3.6: Trust-aware deployment diagram

3.3 A Meta-Model for Context-Patterns

Requirements define what properties and functionality a software should have. It is impossible to assess the quality of a software without requirements. Moreover, writing requirements is only possible if the domain knowledge of the system-to-be and its environment is known and considered thoroughly. We address this problem by describing common structures and stakeholders for several different domains in so-called *context-patterns*. We show several example patterns in Section 2.4.

In this section, we present a meta-model for context-patterns. The meta-model shows a common structure of our context-patterns and supports software and security engineers in describing their own context-patterns. We illustrate this process in an example. This section is a summary of our EuroPlop contribution [8].

3.3.1 A Meta-Model for Context-Patterns

Our meta-model for building context-patterns considers domain knowledge during the analysis phase of software engineering. We consider different kinds of domain knowledge, e.g., technical domain knowledge. Therefore, we use a bottom up approach, starting with a set of previously and independently developed context-patterns.

We identify the common concepts in our existing context-patterns shown in Section 2.4 and aggregate this knowledge into a meta-model of elements. This is quite similar to what Jackson [22] proposed for requirements. He defined a meta-model of reoccurring domains, like causal, biddable and lexical domains. These domains are used to define basic requirements patterns, so-called *Problem Frames*. In this work, we show a similar meta-model for context elicitation. We show *how* we derived it from already existing context-pattern, and how it can be used to describe the structural part of a new context-pattern.

This meta-model has several benefits. First, it forms a uniform basis for our context-patterns, making them comparable. Second, findings and results for one pattern can be transferred to the other pattern via a generalization of meta-model elements. Third, the meta-model contains the important conceptual elements for context-patterns. Fourth, it enables us to form a pattern language for the context-pattern. However, in this work we focus on the aspects of the meta-model, which create the basis of a pattern language for context elicitation.

Using this meta-model we empower requirements and software engineers to describe their own context elicitation patterns, which capture the most important parts for understanding the context of a system-to-be. Note, that the difference between our meta-model and Tolendano's [41] meta-pattern is that he abstracted existing patterns into meta-pattern, while we want to create a meta-model as a basis for a pattern language for context elicitation.

The meta-model was derived in a bottom up way from the different patterns we described independently for different domains. For the process of deriving the general elements, which then form the meta-model, we started to analyse each context elicitation pattern in isolation. For each element in a context elicitation pattern we discussed, what the general concept behind this element is or if it is a general concept in itself. Therefore, we set up a table containing the elements of the current pattern to be analysed as rows and the conceptual elements as columns. For each concept found we checked if this concept was already covered in the table or not. In the case it was already covered we only added a cross to the table. In the case it the concept was not covered by a conceptual element, we added a new column. After iterating over the elements of the pattern, we did a second step by adding the found conceptual elements as rows and analysing for each of them if they could be further generalized in a reasonable way or not. This way we found also new conceptual elements. Hence, we had to do the second step several times. If nothing new was found, we finished the analysis. This way, we obtained the conceptual elements, which were candidates for the meta-model.

Table 3.2 shows the result of this phase for the SOA pattern. In this case, we analysed two pattern in conjunction, because the stakeholder SOA pattern reuses many elements of the SOA layer pattern.

In a next phase we harmonized the conceptual elements by comparing the found elements, merging them if needed and setting up their relations. This way we got a coherent set of conceptual elements over all patterns.

In the last phase we had to choose, which conceptual elements should be part of the meta-model. Table 3.3 shows the conceptual elements and in which of the patterns a corresponding element exists.

		General Concept												
		Layer	Stake-holder	Pro-cess	Active Re-source	Rela-tion	Envi-ron-ment	Indi-rect Envi-ron-ment	Indi-rect Stake-holder	Direct Envi-ron-ment	Direct Stake-holder	Ma-chine	Area	Re-source
SOA Layer Pattern Element	Business Organisations	x												
	Organisation		x											
	Business Processes	x												
	Process			x										
	Business Services	x												
	Business Service				x									
	Infrastructure Services	x												
	Infrastructure Service				x									
	Component-based Service Realization	x												
	Component				x									
	Operational Systems	x												
	CRM				x									
	ERP				x									
	Database				x									
	Packaged Applications				x									
	Legacy Applications				x									
	Participates In					x								
	Performed By					x								
	Relies On					x								
	Exposes					x								
	Business Relation					x								
Stakeholder SOA Pattern Element	Outer System						x							
	Indirect Environment							x						
	Legislator								x					
	Domain								x					
	Shareholder								x					
	Asset Provider								x					
	Inner System						x							
	Direct Environment									x				
	Process Actor										x			
	Business Service Provider										x			
	Infrastructure Service Provider										x			
	Component Provider										x			
	Operational Systems Provider										x			
	Machine											x		
	Influences					x								
	Part Of					x								
	Provides					x								
Conceptual Element	Layer												x	
	Stakeholder													
	Process													
	Active Resource													x
	Relation													
	Environment												x	
	Indirect Environment						x							
	Indirect Stakeholder		x											
	Direct Environment						x							
	Direct Stakeholder		x											
	Machine												x	
	Area													
	Resource													

Table 3.2: Analysis of the SOA Layer Pattern and the Stakeholder SOA Pattern Elements

Additionally, we selected for each pattern those elements which were not explicitly part of the pattern and checked if the missing element is an implicit part of the pattern. The patterns were also tagged with the information if there is a technical or organisational view provided or a combination of both. This is important to consider, because there might be elements, which only occur in one of the views. Those elements might be excluded by just looking at the pure occurrence number, because they can only occur in a subset of the pattern. But those elements might be nevertheless important to capture aspects which are special for a view.

The general rule to include an element into the meta-model or not, was to add every element with an occurrence greater than three, which means the element occurs in more than the half of the patterns. In case of a view specific element an occurrence of greater than two was sufficient, because the number of patterns associated with a view was four. Every element with an occurrence of two was subject to be discussed. The occurrence of an element was calculated only considering the explicit occurrence in a pattern.

	Type	technical	technical	Technical, organisa-tional	technical, organisa-tional	organisa-tional	organisa-tional
	Pattern	P2P Pattern	SOA Layer Pattern	Stake-holder SOA Pattern	Cloud Pattern	Law Iden-tification Pattern	Law Pattern
Meta-model Element	Pattern	x	x	x	x	x	x
	Areas	x	x	x	x	x	x
	Machine	x	x	x	x		
	Environment		x	x	x	x	x
	Direct Environment		x	x	x	x	x
	Indirect Environment		x	x	x	x	x
	Layer	x	x	x			
	Process		x	x		x	x
	Activity		x	x		x	x
	Stakeholder		x	x	x	x	x
	Direct Stakeholder		x	x	x	x	x
	Indirect Stakeholder			x	x	x	x
	Resource	x	x	x	x	x	x
	Active Resource	x	x	x	x		
	Passive Resource				x	x	x
	Relation	x	x	x	x	x	x
uncovered Elements	Requirements	x				x	
	Requirements leading to P2P	x					
	Requirements Influenced by P2P	x					

x = contains element x = contains element implicit

Table 3.3: Overview of Elements of the context-pattern and their relation to the Meta-model

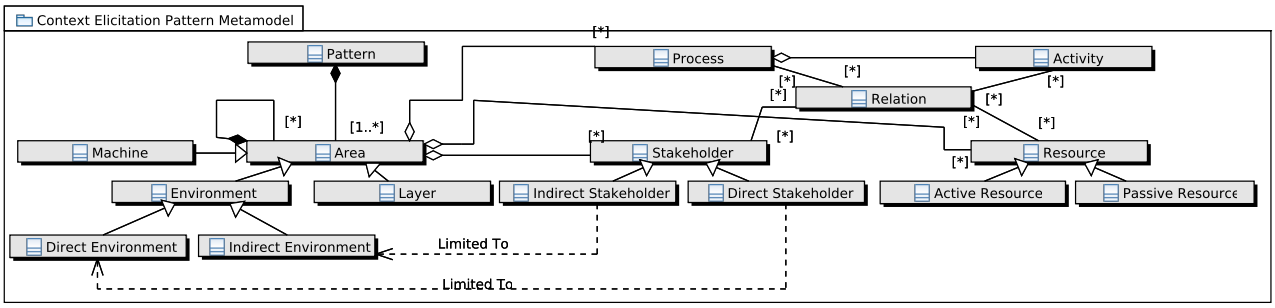

Figure 3.7: context-pattern Meta-model

We had to discuss the conceptual elements requirement and machine. For the machine element it seemed that it is only part of patterns which mix-up the technical and the organisational view. So the first reason to include them is that for eliciting the context of a software problem the most usual pattern is one which mixes the technical and organisational view. This reason was supported by the experiences of the authors and context-patterns, which are currently developed and studied, but which are not published yet. A second reason was the fact that the patterns with a more technical view contain the machine implicitly. For example, for the SOA Layer pattern (see Figure 3.8) the machine is not an explicit model element, but the extension to the Stakeholder SOA pattern (see Figure 3.9) shows that elements of the SOA layer pattern directly relate to the machine. We could not find similar evidence for the requirement element. Moreover, we think that the requirement is part of the phases which follow the context elicitation. For the P2P pattern we only added them for visualization means. Law Identification patterns are used in an iterative way. Thus, they are applied after the ideal context without legal restriction is elicited. Hence, this is a very specific case, which one cannot generalize. As result, the requirement element is excluded and the machine element added to the context elicitation meta-model.

Finally, we formed the meta-model as depicted in Figure 3.7 out of the selected conceptual elements. The meta-model was modelled using the UML notation.

The root element is the `Pattern` itself. Each pattern consists of at least one `Area`. In general, an area contains elements of the same kind, view or level. An area can contain other areas to split it up and make it more fine-grained. An area can be the `Machine`, i.e., the thing to be developed, or an `Environment`, which contains in turn elements that have some kind of relation to the machine, or a `Layer`, which encapsulates elements of the same hierarchy level.

The environment can be further refined. There are elements which directly interact with the machine, captured in the `Direct Environment`. And there are elements which have an influence on the system via elements of the direct environment, captured by the `Indirect Environment`.

An element, which is part of an `Area`, can be a `Process`, a `Stakeholder`, or a `Resource`. A process describes some kind of workflow or sequence of activities. Therefore, it can contain `Activities`. A stakeholder describes a person, a group of persons, or organisational units, which have some kind of influence on the machine. A stakeholder can be refined to a `Direct Stakeholder`, who interacts directly with the machine, and an `Indirect Stakeholder`, who only interacts with direct stakeholders, but has some interest in or influence on the machine. A `Resource` describes some material or immaterial element, which is needed to run the machine or which is processed by the machine and which is not a stakeholder. A resource can be an `Active Resource` with some behaviour or a `Passive Resource` without any behaviour.

This meta-model has several benefits. First, it forms a uniform basis for our context-patterns, making them comparable. If a method already makes use of one of the patterns, it is now easy to generalize the usage to the elements of the meta-model. This enables one to replace a given used pattern by another one easily. Second, findings and results for one pattern can be transferred to the other pattern via a generalization to the meta-model elements. Third, the meta-model contains the important conceptual elements for context elicitation patterns. Thus, it is helpful to know these elements and search for them in a specific domain when setting up a new context-pattern for a domain. Fourth, it enables to form a pattern language for the context elicitation pattern. The common meta-model eases relating the patterns to each

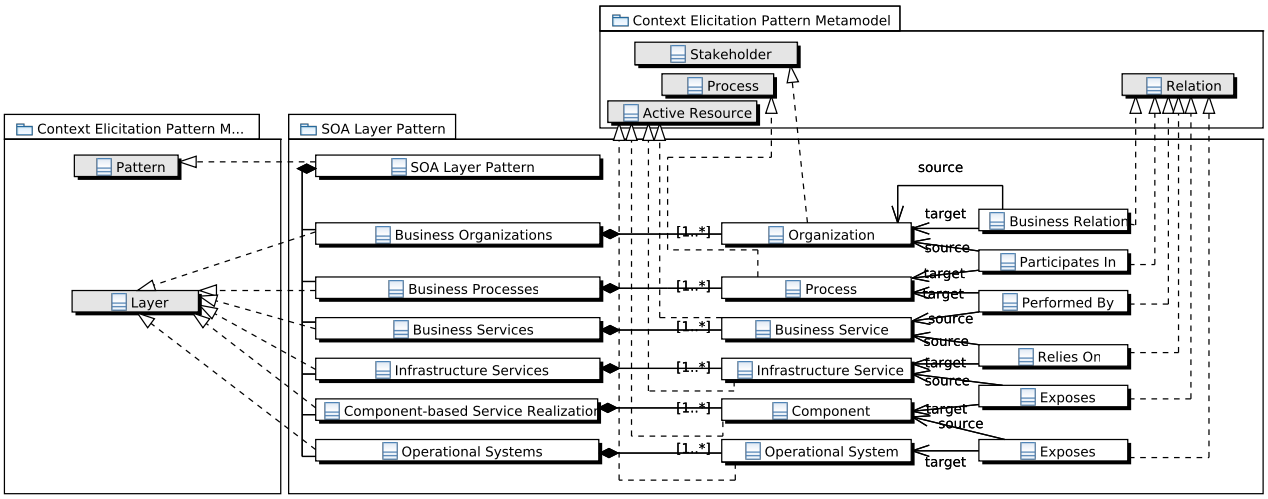

Figure 3.8: SOA Layer Pattern Meta-model

other.

3.3.2 Using the meta-model to describe a context-pattern

After the definition of the meta-model we instantiated it for each of our context-patterns. Thus, we aligned all of the patterns to the same foundation making them comparable. Additionally, when integrating context elicitation patterns into requirements engineering methods this can be done in general only referring to the context elicitation meta-model.

An example of the result of the application of the meta-model can be seen in Figures 3.8 and 3.9. Figure 3.8 shows the meta-model of the SOA layer pattern. The root *pattern* element is the *SOA Layer Pattern* itself. *Business Organisations*, *Business Processes*, *Business Services*, *Infrastructure Services*, *Component-based Service Realization*, and *Operational Systems* are instances of the *Layer* element and part of the *SOA Layer Pattern*. *Organisation* is an instance of the *Stakeholder* element. *Process* is a *Process*. *Business Service*, *Infrastructure Service*, *Component*, and *Operational Service* are instances of the *Active Resource*. *Business Relation*, *Participates in*, *Performed By*, *Relies On*, and *Exposes* are *Relations*. As the stakeholder SOA pattern is an extension of the SOA layer pattern, it re-uses elements of the SOA layer pattern as shown in Figure 3.9. It adds the *Stakeholder SOA Pattern* as new *Pattern* instance. *Environment* instances are the *Outer System* and the *Inner System*. The *Indirect Environment* and the *Direct Environment* of the stakeholder SOA pattern are instances of the *Indirect Environment* respective *Direct Environment* of the context elicitation meta-model. The Stakeholder SOA pattern adds an organisational view to the technological focus of the SOA layer pattern. Hence, there is a *Machine* explicitly. As the pattern is stakeholder centric, it basically adds *Stakeholder* and their *Relations*. *Legislator*, *Domain*, *Shareholder*, and *Asset Provider* are instances of the *Indirect Stakeholder* element. For *Direct Stakeholder* is instantiated as *Process Actor*, *Business Service Provider*, *Infrastructure Service Provider*, *Component Provider*, and *Operational Systems Provider*. The new *Relations* added are *Influences*, *Part Of*, *Participates In*, and *Provides*. These two examples of the application of the meta-model show that the meta-model is sufficient for instantiating context-pattern. Each pattern could be described using the meta-model without any problems. This way we prove that the generalization we did for forming the meta-model was reasonable.

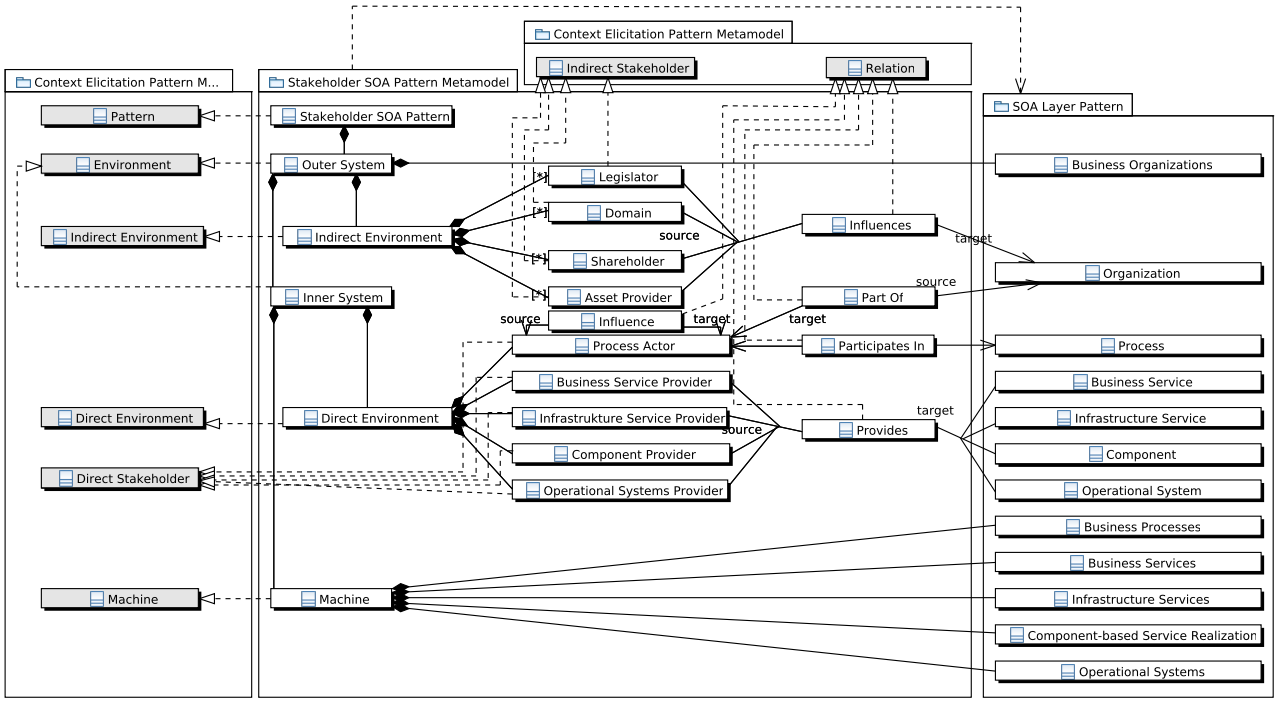

Figure 3.9: Stakeholder SOA Pattern Meta-model

4 Context and Security Requirements Patterns

This section describes reasoning techniques that take advantage on extensions of modelling languages, which are previously discussed in Section 3. Section 4.1 describes our technique for evaluating the security of cloud scenarios using trust relationships. Our technique supports the structured elicitation of relevant trust information and their analysis. The outcome of our work helps decision makers to decide for or against the use of clouds. Section 4.2 provides an overview of the patterns we have designed to detect resource, goal and compliance conflicts. Section 4.3 reports the patterns to detect potential insider threats in a requirements model representing system's stakeholders and the organisational settings.

4.1 Trust-aware Cloud Pattern

Cloud computing (or short: clouds) provides scalable IT resources. Clouds promise cost savings and flexibility for cloud customers, but cloud customers are reluctant to trust the security of clouds in companies. The lack of trust in cloud security is caused by the design of clouds: storing and managing critical data and executing sensitive IT-processes is performed beyond the companies/customers control.

We propose a cloud-specific technique for eliciting and analysing relevant trust relations. The outcome of our technique helps to decide if a cloud scenario should be pursued or not.

Our work is based on an enhanced version of the cloud system analysis pattern. The pattern defines stakeholders, technological artefacts, and their relations. We use the pattern to identify trust relations in the cloud and calculate trust values based on behavioural data of cloud stakeholders. The calculation uses existing methods and the results are used to evaluate cloud scenarios with regard to security, e.g., if a certain IT process can be entrusted to a specific cloud provider. We illustrate our method with an eHealth scenario.

4.1.1 Cloud Pattern

We proposed patterns for a structured domain knowledge elicitation in Section 2.4. Depending on the kind of domain knowledge that we have to elicit for a software engineering process, we always have certain elements that require consideration. For this work we use a specific *context elicitation pattern*, the so-called *cloud system analysis pattern* [11, 7].

We base our approach on Jackson's work on Problem Frames [22] that considers requirements engineering from the point of view of a machine in its environment. The machine is the software to be build and requirements are the effect the machine is supposed to have on the environment. Any given environment considers certain elements, e.g., stakeholders or technical elements. Jackson [22] describes Problem Frames as follows: *"A problem frame is a kind of pattern. It defines an intuitively identifiable problem class in terms of its context and the characteristics of its domains, interfaces and requirement"*.

We were also inspired by Fowler [19], who developed patterns for the analysis phase of a given software engineering process. His patterns describe organisational structures and processes, e.g., accounting, planning, and trading.

Our patterns for the analysis phase differ from patterns concerning solutions for the design phase of software engineering like the Gang of Four patterns [20] or the security patterns by Schumacher et al. [39]. The reason is that we provide a means for a structured elicitation of domain knowledge for cloud computing systems. We do not provide solutions for the implementation phase of clouds.

We present a short introduction of our so-called *Cloud System Analysis Pattern (or short: Cloud Pattern)* [7] in Section 4.1.1. We created the pattern for cloud-specific establishment of an information security management system compliant to the ISO 27001 Standard.

4.1.2 Cloud Stakeholder Templates

We accompany our cloud system analysis pattern by templates to systematically gather domain knowledge about the direct and indirect system environments based on the stakeholders' relations to the cloud and to other stakeholders. We updated the templates with location, cloud deployment scenarios, and privacy concerns with respect to a previous publication [11].

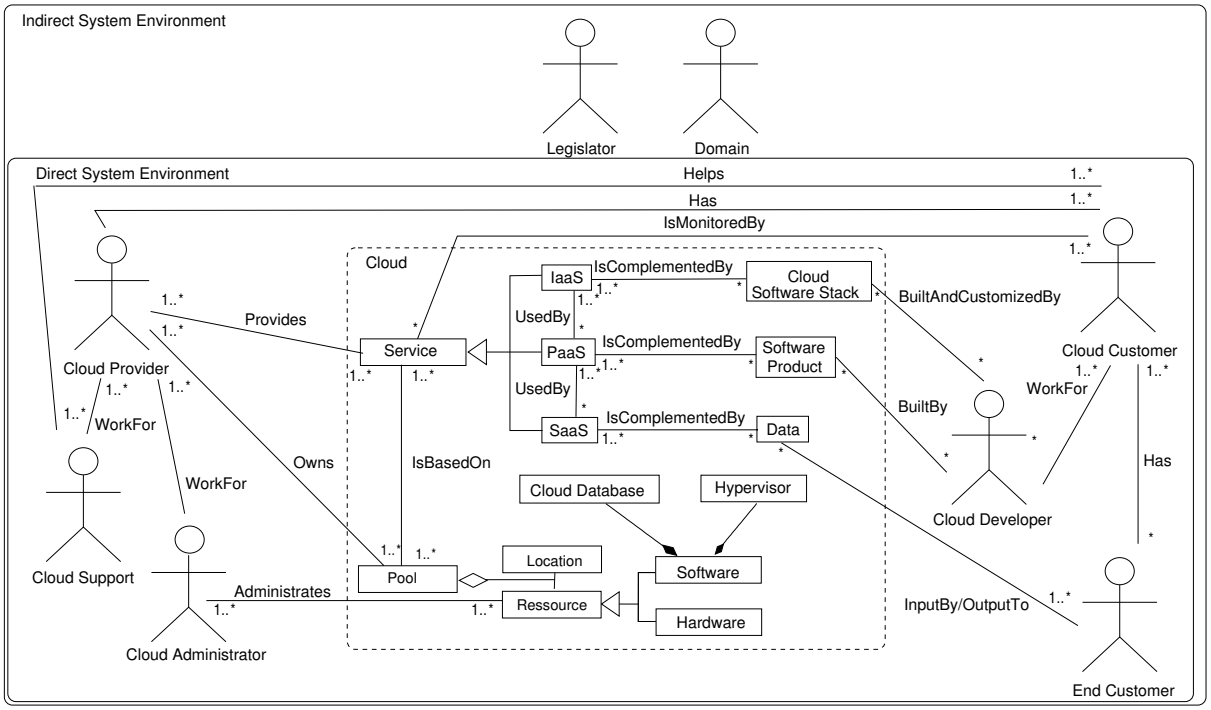

Figure 4.1: Extended Cloud Computing Pattern taken from [7]

The first template serves to describe stakeholders contained in the direct system environment, shown in Table 4.1. The second template describes the stakeholders contained in the indirect system environment (see Table 4.2).

Table 4.1: Direct Stakeholder Template - updated version from [11]

Name	State the identifier of the stakeholder or group of stakeholders, e.g. company name or group of end customers.
Description	Describe the stakeholder informally, e.g., if the stakeholder is a natural or a legal person.
Relations to the cloud	Describe the input and output represented as relation (line from this stakeholder to the cloud) between the stakeholder and the cloud, e.g., the kind of data or software.
Cloud deployment scenarios	State the deployment scenarios the cloud stakeholder demands: public, private or hybrid. Also state the reason for the particular deployment scenario.
Location	State the country the stakeholder works in.
Motivation	State the motivation of the stakeholder for using the cloud based on the previous considered relations to the cloud, e.g., business goals such as profit increase.
Relations to other direct stakeholders	For each relation (line from this stakeholder to another direct stakeholder), name the kind of dependency between the stakeholders, e.g., indirectly influenced by customer-demand.
Assets	State the assets of the stakeholder that are already known.
Compliance	State relevant laws and regulations for the cloud scenario that are already known.

In addition to our method, we have a hierarchical structure of models, which lets us analyse the cloud system at different decomposition levels or views.

4.1.3 Trust Background

There has been a huge number of definitions of trust over the years. This is due to mainly two factors: first, trust is very context-dependent, and each context has its own particularities. Second, trust spans across many disciplines, including psychology, economics and law, and have different connotations in each of them. Finally, there are many factors that influence on trust, and it is not straightforward to identify them all.

Table 4.2: Indirect Stakeholder Template - updated version from [11]

Name	See direct stakeholder template.
Description	See direct stakeholder template.
Relations to other stakeholders	For each relation from this stakeholder to another direct or indirect stakeholder (no line explicitly shown), name the kind of dependency between the stakeholders, e.g., protected by, controlled by law, implement laws.
Motivation	State the motivation of the stakeholder for having any reason of considering the cloud for its work or the motivation for having any kind of relation to stakeholders of the direct or indirect environment, e.g., protect privacy of citizens or implement concrete laws of an economic community.
Compliance	Identify relevant laws as well as regulations based on the indirect stakeholders. Specify and identify the ones relevant for the stakeholder at hand, e.g., HIPAA.

If we consider some of the trust definitions provided in the literature over the last years, we come up with certain concepts that repeat in most of them. In Moyano et al. [30], we gathered, structured and studied the relation between these concepts. Some important notions related to trust are entities, risk, uncertainty, context and security.

As we are dealing with a cloud scenario and this is quite attached to the notion of service, we stick with the definition by Olmedilla et al. [33]: 'trust of a party A to a party B for a service X is the measurable belief of A in that B behaves dependably for a specified period within a specified context (in relation to service X)'.

As discussed by Moyano et al. [30], there are different types of trust models. For the purpose of this work, we are interested in evaluation models. In these models, the factors that have an influence on trust are identified, quantified and aggregated to yield a trust score. Figure 4.2 and Figure 4.3 show the most important concepts that are related to these models.

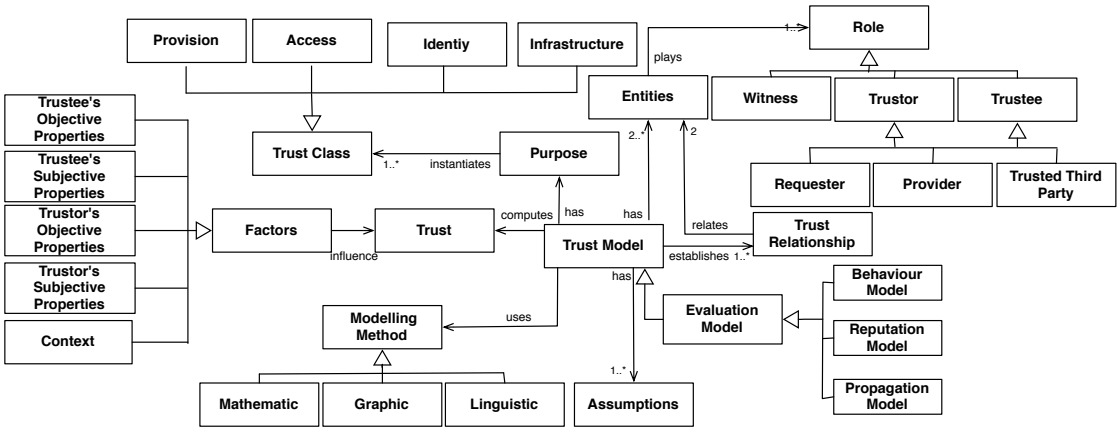

Figure 4.2: Trust Common Concepts

The notion of trust builds upon the notion of trust relationship. A trust relationship expresses how much a trustor trusts a trustee by means of a trust value. This trust value may be influence by many factors, including trustor's subjective and objective factors and trustee's subjective and objective factors. The context is also crucial in determining this trust value.

In order to quantify these factors, we may use different sources of information, including reputation, first- and second-hand experiences or social information (e.g., the roles being played by the actors), among others. Evaluation models differ in the types of factors that take into consideration when computing trust, as well as in the way to retrieve information of these factors.

We can also define subjective factors for non-human entities, such as components or nodes. Trust computation among these entities can be approached in two different but non-exclusive ways. First, it is possible to consider these entities as agents capable of holding beliefs and a mental state based on prejudice or previous experiences. Second, Quality of Service (QoS) and Service Level Agreements (SLAs) can provide good indicators to determine levels of trust.

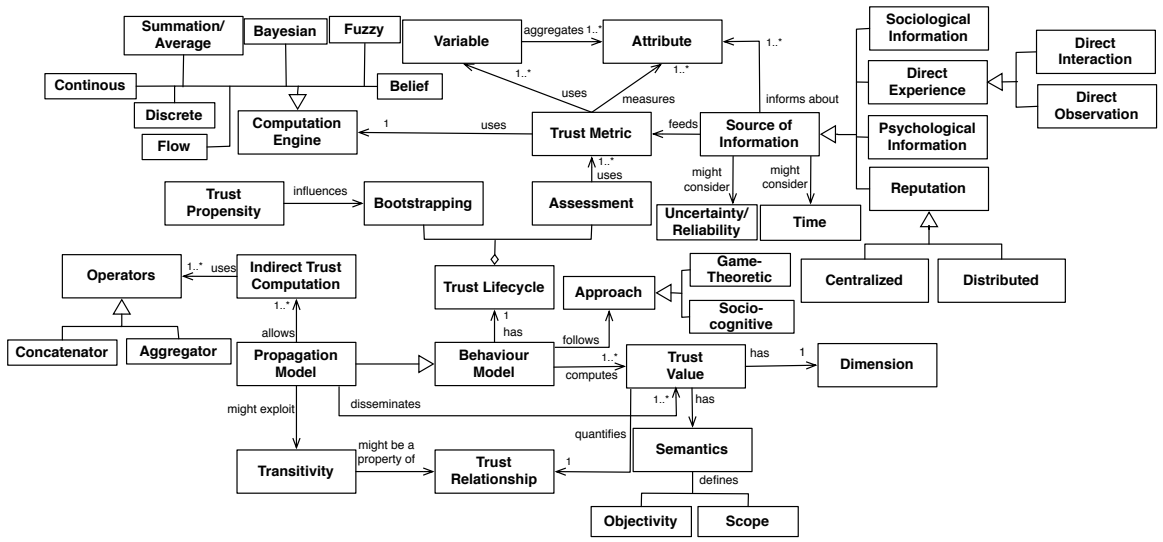

Figure 4.3: Evaluation concepts

The idea is therefore, extending the trust model with some of the previous concepts so as to allow a trust analysis in the cloud pattern. This is further explained in the next section.

4.1.4 Extending the Cloud Pattern

We improved our cloud system analysis pattern, depicted in Figure 4.4. We included the direct stakeholder *Administrator*, who customizes the *Cloud Software Stack*. In addition, we added *Service*s at the *PaaS* and *SaaS* level of the cloud. These services are not provided by the *Cloud Provider*, but by a service provider. The *Cloud Developer* integrates *Service*s in order to build the *Software Product*. The *End Customer* uses Services that can complement the functionality of the clouds *SaaS* offers.

Trust and Reputation Template

In order to evaluate this trust, we can collect information using our trust and reputation template (see Table 4.3). This information will be used as input to a trust metric or trust evaluation procedure, the output of which will be the trust value between the delegator and the delegate.

Service Template

We use a service template to gather the knowledge regarding the services that both cloud developers and end consumers can use.

Application to the eHealth Case Study

We instantiate our cloud pattern using the example of the eHealth scenario regarding electronic health records (EHR)s, depicted in Figure 4.5. The hospital is located in Germany. The *Hospital* offers a service to manage *EHRs* in a cloud computing environment. The hospital orders the development of the *Online EHR Service* by the *Internal Development Unit*. The cloud provider *Hulda* provides the cloud resources to implement the service. The *Patient stores/changes medical data*. The *VIP Patient* stores/changes also service request in the medical data. The *Patient* and the *VIP Patient* have the same relations to the cloud. For simplicity's sake, we do not show the *VIP Patient* in the pattern. However, the stakeholders have different instantiations of stakeholder templates. A specific service request is the so-called *Additional Treatment Request* that allows the VIP patient to acquire additional medical services or environmental request like a room with a view towards the hospital garden.

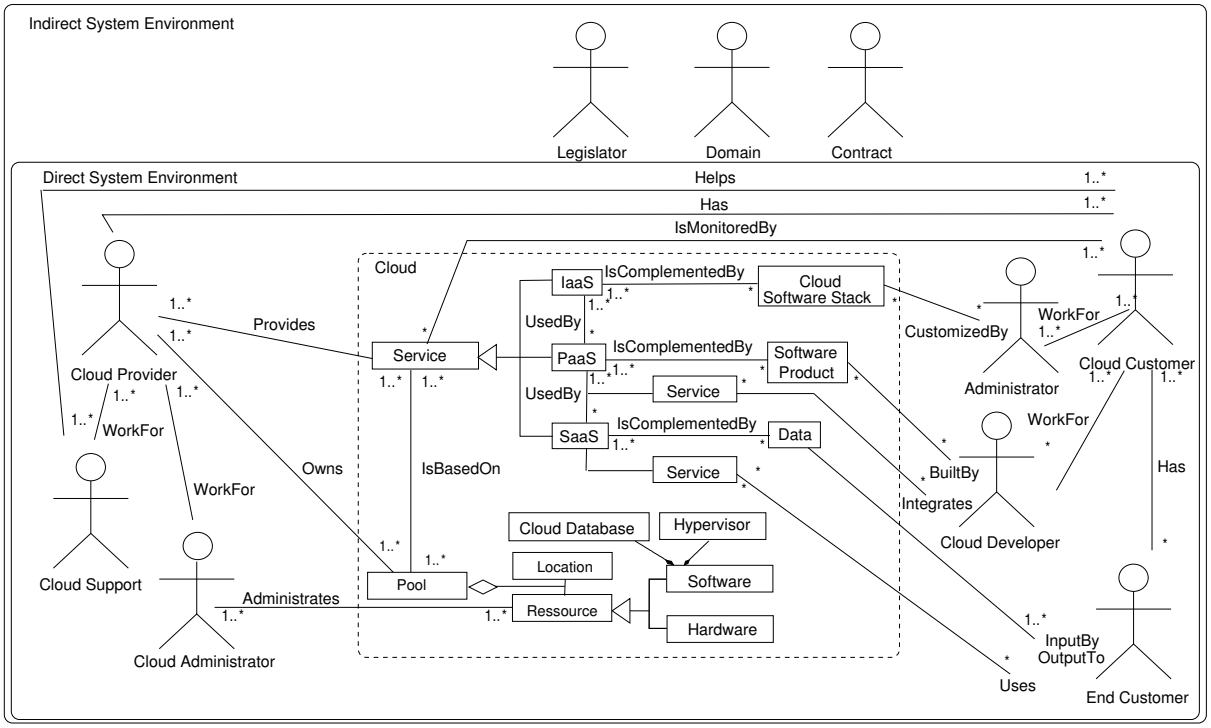

Figure 4.4: Extended Cloud Computing Pattern

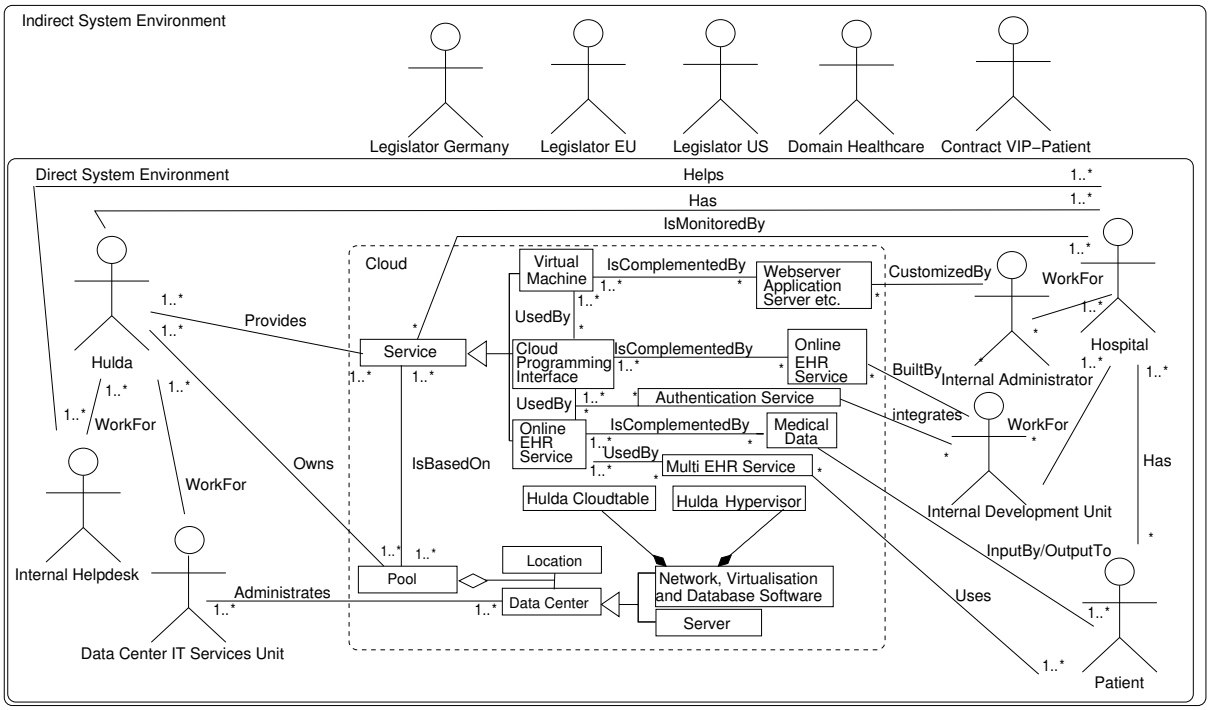

Figure 4.5: Instantiated Extended Cloud Computing Pattern with an online healthcare scenario

The hospital plans to hire an internal software development unit to develop software for EHR management in the cloud and a customized operating system (OS) for the developed EHR management software. Hence, the hospital plans to outsource the affected IT processes to the cloud to reduce costs and scale up their system for a larger amount of patients. EHRs are stored in the cloud database, and transactions

Table 4.3: Trust and Reputation Template

Information	This refers to a particular information of a cloud stakeholder or a cloud element
Contracts / obligations	Refer to signed contracts that reflect the conditions and liabilities regarding a given task. They provide trust as the trustor (delegator) is protected by the law, since he knows that the delegatee will be punished should something goes wrong.
Previous Experience on Task	State the delegatee's previous experience on a task of the same class or just on similar tasks could be a good indicator that the delegatee will perform well.
Previous Direct Interactions	State if the delegator has had previous interactions with the delegatee, we can use the outcome (successful/failure) of these previous interactions. (e.g., the delegator delegated a different task to the delegatee and the latter performed it well).
Previous Indirect Interactions	State if the delegator may delegate the task to another stakeholder who, in turn, can delegate the task to the delegatee. If the task is well performed, the delegator can conclude that the delegatee performs well in spite of not having experienced a direct interaction.
Previous Direct Observations	State previous Direct Observations: the delegator may use the observations made during previous/current interactions between the delegatee and any other entity/stakeholder.
Previous Indirect Observations	State previous Indirect Observations: the delegator can consider the observations made by other stakeholders about interactions of the delegatee with any other stakeholders.
Reputation	State if the delegatee has a reputation, which aggregates the opinions of the rest of stakeholders in the Cloud setting. It requires a long record of interactions.
Membership to a group	State if the delegatee belongs to a group with a high or low reputation, inheriting the reputation of this group.
Role within the organisation	State if the delegatee plays one or several roles in the organisation. This role can determine at a certain extent how well the delegatee will perform the task.
Willingness to the Task	Consider the level of motivation that the delegatee presents to do the task. State if she/he is fond of the task and consequently likely to perform better than if she/he hates it.
Knowledge on the Task	State if the delegatee has the required knowledge/credentials/preparation to perform well.

Table 4.4: Service Template

Name	State the identifier of the service
Informal Description	Describe the goal of the service in an informal way
Location	State where the service is geographically hosted.
Deployment environment	State the application server onto which the service is deployed.
Preconditions	State the conditions that must be true prior to using the service.
Postconditions	State the conditions that must be true after using the service.
Input information	Indicate what information the service requires as input.
Output information	State the outcome produced by the service.
Atomicity	State whether the service is atomic or a result of a service composition.
Ownership	Indicate who is the owner of the service. It could be an individual, a company or it may be diluted due to complex compositions from different organisations or individuals.
Properties/Non-Functional requirements	Properties that the service hold e.g., response time < x seconds.
Compliance	State whether the service is compliant with certain laws or regulations (e.g., laws for protection of personal private information).
Testing/Assurance	Indicate whether the service has been through some testing/assurance method.

like transfers of medical information from the hospital to a local physician are processed in the cloud.

The internal development unit creates SaaS systems for the hospital via developing in a PaaS environment. The internal development unit also integrates the *Authentication Service* from an external source, which authenticates cloud customers for the online banking service. This authentication services is offered for numerous cloud services from Hulda and other cloud providers. Hence, allowing a single-sign-on functionality, when a cloud customer only uses services that use the authentication service.

The internal administrator customizes an OS for a given IaaS offer and installs a web server, application server and further more. The bank authorizes its internal software department, to design and build the cloud-specific software according to the interface and platform specification of the cloud provider.

The main goal of the cloud provider, in our example a company called *Hulda*, is to maximize profit by maximizing the workload of the cloud. Therefore subgoals are to increase the number of customers and their usage of the cloud, i.e., the amount of data as well as the number and frequency of calculation activities they outsource into the cloud. Fulfilling security requirements is only an indirect goal to acquire customers and convince them to increase the subset of processes they outsource.

The patient is a person, juristic or natural, who uses the EHR services offered by the hospital, which

allows the patient to view his/her medical data, request procedures, or transfer medical data to other care providers.

In our scenario, the transactions concerning medical data can be done via the web service the hospital offers using the cloud. The patient uses also a *Multi EHR Service*, which collects all transactions from all EHR of all care providers the patient has. Hence, the multi EHR service provides an overview of the overall medical situation of the patient.

Basically, the online EHR cloud service is embedded in an environment consisting of two parts, namely the *Direct System Environment* and the *Indirect System Environment*. The *Direct System Environment* contains stakeholders and other systems that directly interact with the cloud through associations, e.g., the *Patient*. Moreover, associations between stakeholders in the *Direct* and *Indirect System Environment* exist, but not between stakeholders in the *Indirect System Environment* and the cloud. For example, the *Legislator Germany* is part of the *Indirect System Environment*. Typically, the *Indirect System Environment* is a significant source for compliance and privacy requirements.

We derive the indirect stakeholders required for this scenario based on the instantiation of the *Direct System Environment*. The *Cloud* is located in Germany and the USA. This is the reason for the *indirect stakeholders Legislator* Germany and *Legislator* US. Germany is a member of the European Union resulting in an additional set of regulations. They are described by the *Legislator* EU that represents a set of EU regulations.

The hospital has also several contractual obligations, one of which is the *Contract VIP-Patient* that defines the services for health and comfort the hospital offers to its VIP patients. As examples, we present one stakeholder template instance for an indirect stakeholder (see Table 4.5) and one for a direct stakeholder (see Table 4.6).

Table 4.5: Indirect Stakeholder Template: Legislator Germany (cf. [11])

Name	*Legislator Germany*
Description	The *Legislator Germany* represents all German laws relevant for this cloud scenario.
Motivation	The German laws try to control the risks of companies (*Hulda* and *Hospital*) and to protect the privacy of the *Patients* by regulating disclosure of personal data.
Relations to other stakeholders	Controlled by law: The laws have to be obeyed by all stakeholders of the *Direct System Environment*.
Compliance	The following regulations might be considered: • Privacy protection: e.g., BDSG • Risk management: e.g., AktG

Table 4.6: Direct Stakeholder Template: Patient

Name	*Patient*
Description	The *Patient* uses the online EHR service of the *Hospital* and the multi EHR service of a third party.
Motivation	The *Patient* wants easy and secure medical data transactions via the hospital‘s cloud computing offer.
Relations to the cloud	*InputBy/OutputTo*: *InputBy medical data, data related to a person*, which is required for treatment of the *Patient* by the *Hospital*.
Cloud deployment scenarios	The hospital considers using a public cloud, because it offers significant savings in terms of money. The *Patient* hopes for a subsequent cost reduction for his/her treatment.
Location	The *Patient* is located in Germany.
Relations to other direct stakeholders	*Has*: *Hospital* as SaaS provider
Assets	Medical data and all data related to the person
Compliance	The following laws might be of relevance: • Privacy protection: BDSG Section 3, Section 4, Section 9, Section 11 • Risk management: AktG Section 91, Section 93

Table 4.7: Service Template: Authentication Service

Name	Authentication Service
Informal Description	The service authenticates end customers for uses of SaaS offers.
Location	The service is hosted in the U.S.
Deployment environment	The service is deployed on a server developed by the owning company ThirdPartyTrust. The server relies upon encryption protocols like SSL for communication with all outside sources.
Preconditions	The service requires that end customers have an existing user account with the ThirdPartyTrust Authentication Service.
Postconditions	The service will not provide any information about the end customer to the using PaaS service e.g., name or birthdate. The PaaS service simply get the information that an end user is authenticated successfully or not.
Input information	The service requires as input the end user credentials and the PaaS service identifier the user wants to use.
Output information	The service produces an accept/reject message to the requesting service as an output.
Atomicity	This service is not composed of another service.
Ownership	The owner of the service is the ThirdPartyTrust company, which has its headquarters in the U.S.
Properties/Non-Functional requirements	The service promises an availability of 99.999 percent of the time and a response time < 2 seconds.
Compliance	The service is compliant with the safe harbour agreement between the U.S. and the EC.
Testing/Assurance	The service has been tested according to the Vulnerability Assessment (AVA) specified in the Common Criteria.

Table 4.8: Service Template: Multi EHR Service

Name	Multi EHR Service
Informal Description	The service connects to multiple EHR services in order to provide a medical overview for patients. The service can also conduct medical data transactions using the different services.
Location	The service is hosted in Spain.
Deployment environment	The service is deployed on a tomcat application server.
Preconditions	The user has to be authenticated using the Authentication Service prior to using the service. The user also has to provide the authentication information to the EHR services.
Postconditions	The service keeps all transaction information and medical data of the patient confidential. The service also checks the integrity of all transaction information and medical data between all EHR services.
Input information	The service has to provide the accounts the multi EHR Service shall use and the information of which transactions shall be conducted and which medical data shall be stored.
Output information	The service provides the results of medical data transactions and inquires, and the updated medical data.
Atomicity	The service uses the Authentication Service.
Ownership	This services is owned by the Health Software Solutions company, which has its headquarters in Spain.
Properties/Non-Functional requirements	The service promises an availability of 99.9999 percent of the time and a response time < 4 seconds.
Compliance	The service is compliant to the data protection regulations of the EC.
Testing/Assurance	The service has been tested using mutation-based fuzzing techniques.

Evaluation Trust Model

Figure 4.6 shows a high-level view of a generic trust model. The Trust Relation Template (TRT) is used as an input to the trust model and the output is a qualitative value to tag the corresponding trust relationship. Figure 4.7 shows a decomposition of the trust model black box in lower-level units or activities that follow the process from receiving a TRT to the final trust value output.

Figure 4.6: High-level Generic Trust Model

The first step consists of specifying the relevance of each entry of the TRT. Relevance is determined by means of weights. This can be done in parallel with another activity, which is to assign values to the content of each entry in the TRT. These two activities will heavily depend on the scenario and on the knowledge of the analysts and requirements engineers. It is important to consider the domain and

Table 4.9: Trust Relation Template: Hulda to the Hospital (cloud provider to cloud customer)

Information	Provision of Virtual Machine and the Pool for running an online EHR service
Contracts / obligations	cloud service provision contract with the cloud customer
Previous Experience on Task	Hulda has generally good experience with the hospital. They always pay their used services on time and the required resources of the pool only increase slightly every month.
Previous Direct Interactions	Hulda acquired information about what kind of software the hospital runs on the cloud in order to check compliance obligations. The hospital complied promptly with a functional specification of the online EHR service.
Previous Indirect Interactions	The stakeholders had no indirect interactions.
Previous Direct Observations	Hulda observed that all tasks in the cloud are executed with caution.
Previous Indirect Observations	The hospital was never reported by any other cloud customer for unfair behaviour e.g., excessive use of cloud resources.
Reputation	The reputation of the hospital is excellent. Hulda has never received reports of negative experiences with this cloud customer.
Membership to a group	Hulda is an IT service provider.
Role within the organisation	Hulda belongs to larger organisation that sells hard- and software, as well as logistics services.
Willingness to the Task	Hulda wants to provide cloud services in order to earn money.
Knowledge on the Task	Hulda has a five year experience as a cloud provider.

Table 4.10: Trust Relation Template: Hospital to Patient (cloud customer to end customer)

Information	Transaction information in the online EHR service
Contracts / obligations	The patient has a contract for medical treatment at the hospital.
Previous Experience on Task	The patient has used the online EHR service multiple times and experienced only one minor mistake in one transaction.
Previous Direct Interactions	The patient contacted the hospital to solve issues with an incorrect transaction. The hospital solved the issues within an hour.
Previous Indirect Interactions	The patient interacted with other patients, who had been treated by the hospital. The other patients promised that transactions would never fail.
Previous Direct Observations	The patient experienced one failed transaction and the hospital solved the issues.
Previous Indirect Observations	The patient heard from other patients that some transactions failed or were talking a whole day for processing.
Reputation	The reputation of the hospital was excellent in the beginning of the relation to the patient, but this opinion changed over time, due to ongoing reports of transaction times and failures.
Membership to a group	The hospital is part of a consortium of several hospitals.
Role within the organisation	The hospital is an experimental company for gaining experience with cloud-based online EHR services for the consortium.
Willingness to the Task	The hospital wants to perform the task well, in order to gain good reputation, and to proof to the consortium that cloud-based online EHR is secure and profitable.
Knowledge on the Task	The hospital conducts cloud-based online EHR services for two years.

Table 4.11: Trust Relation Template: Patient to Hulda (end customer to cloud provider)

Information	Hulda stores the patients' medial transaction data and medical data, as well as the information of how often the customer access the online EHR services and for how long. In addition, Hulda knows, which other online EHR services the patient uses and also the access frequency and durations.
Contracts / obligations	Hulda and the patient do not have a contract or obligations to each other.
Previous Experience on Task	The patient has experiences that the availability of Hulda's cloud is excellent.
Previous Direct Interactions	Hulda and the patient have no direct interaction.
Previous Indirect Interactions	The patient received a notification once from Hulda stating that the cloud is offline for 2 minutes. Hulda informed the customer well ahead of time and was offline for online 1 minute.
Previous Direct Observations	Hulda and the patient have no direct observations.
Previous Indirect Observations	The patient observed that the online EHR service conducted every transaction within 3 seconds.
Reputation	The patient has also heard from other users of Hulda's cloud that the availability is excellent.
Membership to a group	The patient is being treated by the hospital, who uses cloud-based online EHR services.
Role within the organisation	The patient is not part of an organisation.
Willingness to the Task	The patient needs to conduct medical data transactions to receive treatment and he/she needs the medical data to have on overview of their health situation.
Knowledge on the Task	The patient is very familiar on how initiate medical data transactions and to look at medical data using the cloud-based online EHR service.

constraints on these values. For the example in Table 4.12, constraints could be: $w_1, ..., w_n \in [0..1]$, $w_1 + ... + w_n = 1$ and $c_i \in (0, 1, 2, 3)$. Domain and constraints influence the following activity: designing the trust engine.

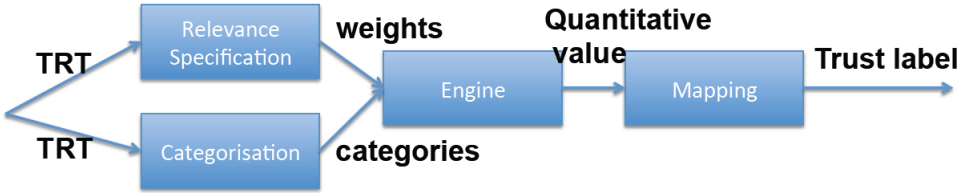

Figure 4.7: Activities carried out by the Trust Model

Table 4.12: w_i refers to weight for that entry; c_i refers to a category value.

Contracts / obligations	w_1	c_1
Previous Experience on Task	w_2	c_2
...	...	...

The trust engine is in charge of aggregating the previous information in order to yield a quantitative trust score. Also, in parallel, we can define a mapping scheme, which basically associates a numeric interval to a numeric value. This mapping and the labels chosen depend on the range of the engine. For example, let us assume that we design the engine to be $v = \sum_{i=1}^{n} w_i c_i$ and we keep the aforementioned constraints. Our engine would have the range $[0, 3]$ and a possible mapping is shown in Table 4.13.

Table 4.13: Example of a Mapping from Quantitative Values to Trust Labels.

$[0, 1)$	Low trust
$[1, 2)$	Neutral trust
$[2, 2.5)$	Medium trust
$[2.5, 3)$	High trust

The final activity consists of converting the numeric value yielded by the engine in a qualitative, more human-readable value by using the mapping defined.

Compositional Trust: Propagation Trust Model

In the previous section, we started off without prior trust information, but a relevant option is to use already-existing trust relationships to infer new relationships. This is the goal of propagation models.

The high-level view of this model is shown in Figure 4.8. The model receives trust values from several trust relationships and yields a new trust value. Depending on the configuration of the trust relationships, we may encounter three situations: aggregation, transitiveness and a combination of them, as depicted in Figure 4.9.

Figure 4.8: Generic Propagation Model

In the examples, we observe patients that want to know how much they can trust a physician based on other patients' relationships.

The aggregation configuration happens when the patient knows how much other patients trust the physician, but he has no trust relationships with such patients. A general constraint that we can impose on this configuration is that the new value cannot be higher than the existing ones, that is, $v_4 \leq v_i, 1 \leq i \leq 3$.

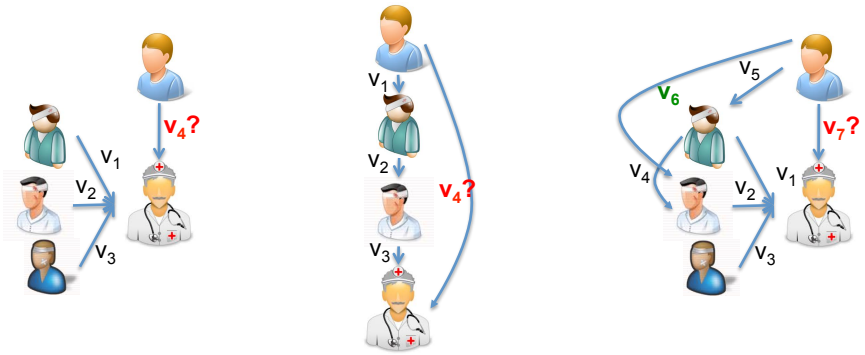

Figure 4.9: From Left to Right: Aggregation, Transitiveness and Combined Configurations

In transitiveness configuration, we have a trust chain among patients. The patient wants to infer a trust value for a physician based on the trust values along a chain of patients. In this configuration, we can impose at least two constraints. First, the final value cannot be higher than the value between the patient and his direct trustee (i.e., $v_4 \leq v_1$). Second, we stop considering values (i.e., feeding the engine with values) as soon as one of them is below a given threshold, which is to be determined according to the range of the engine.

When we have both configurations, the model can proceed as follows: first, it determines values using the chain (v_6 in the example); second, it determines the final value (v_7) by weighting v_1 and v_2 with v_5 and v_6. Another constraint could be that v_1 and v_2 should count more than v_3 as there is no direct trust information about those two patients.

All the aforementioned activities can be performed to assess trust among stakeholders and services, where the Service Template would be another input of the trust model.

4.2 Patterns for Requirements Conflicts

In this section we report the patterns we designed to automatically detect resource, goal and compliance conflicts. The requirements model is modelled by the Si* language.

4.2.1 Patterns for Resource Conflict Detection

The pattern illustrated in Figure 4.10 captures a situation in which a resource conflict arises because

- a resource is linked by a *means end* relation to a goal/task provided by different agents and
- the number of agents who consume the resource to fulfil their goals/tasks exceeds the number of users that can use the resource at the same time.

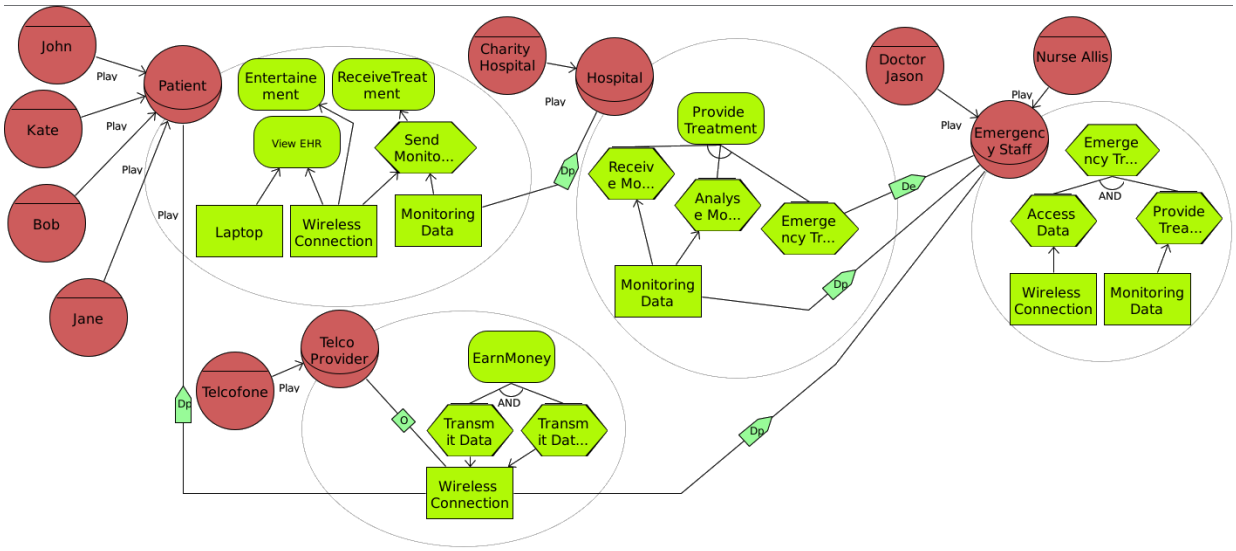

Figure 4.10: An Example of Resource Conflict

In Figure 4.10, the conflicts occur for the limited resource *Wireless Connection* that is provided by the role *Telco Provider* to the *Patient*, *Hospital*, and *Emergency Staff* role. The *Patient* role can be played by the agents *Jane*, *John*, *Bob*, and *Kate*. The *Hospital* role can be played by *Charity Hospital* agent. The *Telco Provider* role is played by *Telcofone* agent. The *Emergency Staff* role is played by *Doctor Jason* and *Nurse Allis*. The *Wireless Connection* is a limited resource because only two agents can use the resource at the same time[1]. The conflict arises because the *Wireless Connection* is provided to roles *Patient*, *Hospital*, and *Emergency Staff* which can be played by three, one, and two agents respectively, and thus the total number of users who use the *Wireless Connection* at same time is six which is greater than the constraint imposed on the usage of the *Wireless Connection*.

4.2.2 Patterns for Goal Conflict Detection

A goal conflict is a situation in which the personal goal of an agent is in contradiction with the goals of the role it plays or when a security goal of an actor collides with the goal of another actor.

Figure 4.11 (without the arrows in the rectangle) shows an example of a Si* model, where a conflict occurs between the main goal of *Jane*, an agent playing the role of the *Patient*, and *Bob* playing the *Doctor*'s role. The conflict arises between the goal *Being Treated* of *Jane* and goal *Earn Money* of *Bob*. *Jane* has delegated the execution of goal *Being Treated* to the *Hospital*. The goal is decomposed in three subgoals *Monitor Patient*, *Manage Patient Data*, and *Diagnose*. The fulfilment of *Diagnose* is delegated to *Doctor* role that is played by *Bob*. However, *Bob* wants to *Earn Money* by enlisting *Jane* to participate

[1] This small number is used for illustration purposes in order to be able to use a simple model with only few actors.

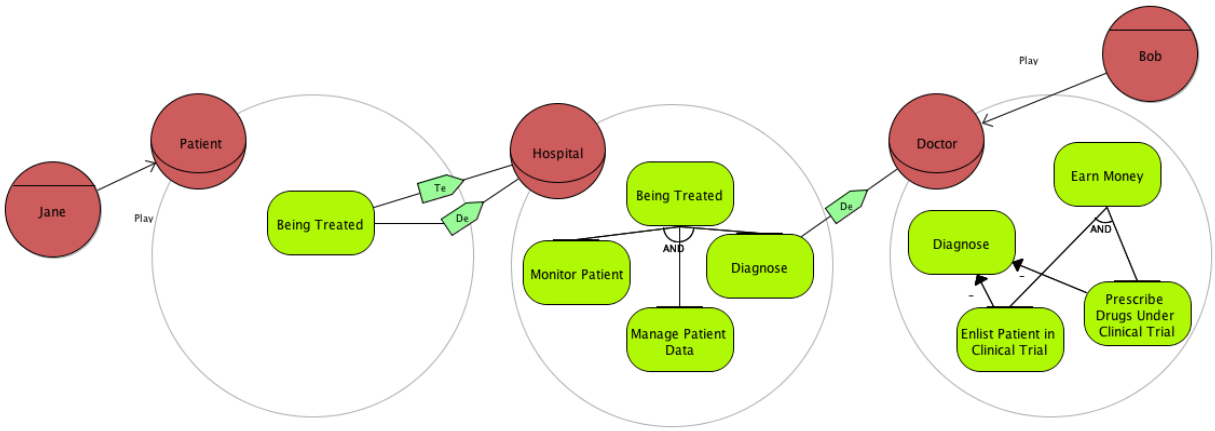

Figure 4.11: An Example of Goal Conflict

in a drug company sponsored clinical trial without her being informed and by prescribing her only drugs under clinical trial. Thus, the subgoals *Enlist Patient in Clinical Trial* and *Prescribe Drugs under Trial* of *Earn Money* wanted by "Bob" negatively contribute to the goal *Diagnose*, and indirectly also to its top goal *Being Treated*.

4.2.3 Patterns for Compliance Conflict Detection

A compliance conflict arises when the requirement model does not conform to the consent expressed by a data subject, or the constraints on the data collection and disclosure imposed by the existing regulations on privacy and data protection. A number of guidelines and regulations for data protection and privacy are available [16, 17, 32], which prescribe how individual sensitive data should be collected, stored and processed. The *Fair Information Practice Principles*[2] (or short FIPs) are widely accepted, which state that a person's informed consent is required before the data is collected and the collection should be limited to the task it is required for and erased as soon as this is not the case anymore. The collector of the data shall keep the data secure and shall be held accountable for any violation of these principles. In the European Union, the *EU Data Protection Directive 95/46/EC* [16] does not permit processing personal data at all, except when a specific legal basis explicitly allows it or when the individuals concerned consented prior to the data processing.

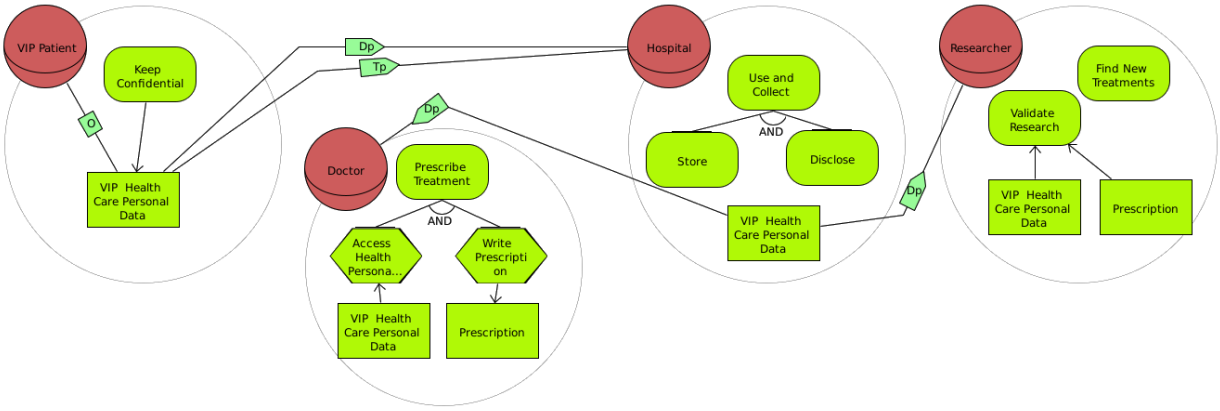

Figure 4.12: An Example of Si* model where compliance conflict occurs

We are able to automatically detect compliance conflicts that arise, because a Si* model is not compliant with the informed consent signed by a data subject. In particular, we present here an example of

[2] http://www.ftc.gov/reports/privacy3/fairinfo.shtm

a compliance conflict where data subject's information is disclosed to a recipient that is not listed in the informed consent signed by the data subject. We illustrate such kind of conflict using the Si* model shown in Figure 4.12, which is related to the management of patients' health care records. In the model, there are four actors *VIP Patient*, *Hospital*, *Doctor*, and *Researcher*. The *VIP Patient* owns his/her *VIP Healthcare Personal Data* and delegates to the *Hospital* the permission to access it so that the *Hospital* can fulfil the goal *Provide Treatment*. However, the *Hospital* further delegates the permission to access *VIP Healthcare Personal Data* to the *Doctor* and the *Researcher*. The *Doctor* requires access to *VIP Healthcare Personal Data* to fulfil its goal *Prescribe Treatment*, and the *Researcher* needs access to *VIP Healthcare Personal Data* to fulfil the goal to validate his/her research. The model violates the informed consent of the *VIP Patient*, which is represented by the Si* model in Figure 4.13. The Figure shows that the *VIP Patient* wants to keep his/her resource *VIP Healthcare Personal Data* confidential, and the *VIP Patient* only authorizes the *Family Doctor*, *Emergency Staff*, *Physician* and *Hospital* to access *VIP Healthcare Personal Data*.

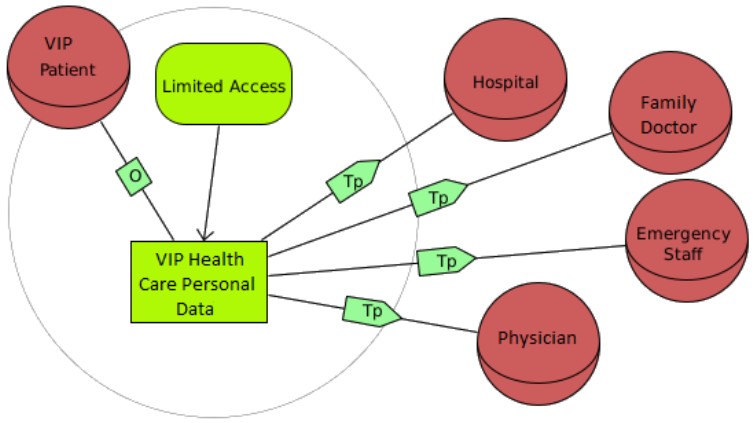

Figure 4.13: An Example of Pattern to model Data Subject's Consent

In order to allow the automatic detection of the patterns in a requirements model, the patterns are specified in the EMF-INCQUERY language. We remind readers to D6.3 [1] for the formalization of the patterns in the EMF-INCQUERY language.

4.3 Patterns for Insider Threat Detection

In this section we report the patterns we designed to automatically detect potential insider threats in a requirements model representing system's stakeholders and the organisational settings. The requirements model is modelled in the extended version of Si* language introduced in Section 3.1.

Since the term "threat" is ambiguous and may have several different meanings, we employ the definition "*A threat is a potential cause of an unwanted incident*" from [28, chapter 4]. A threat is thus an initiator of an unwanted incident (i.e., bad event), not the unwanted incident itself. It is also important to note that a threat does not have to be a human being with malicious intent; a person with good intention making mistakes may also be represented as a threat.

The patterns capture situations where an agent A is an insider threat for a given asset S with risk RL because:

a) A is granted a permission PT on the asset S that is sufficient to violate the security property SP associated with S which has sensitivity level SL and

b) the agent who owns the resource S assigns to A permission PT with a medium-low trust level TL.

As we have explained in Section 3.1, the risk RL associated with an insider threat is given by the sensitivity level SL of the asset S and the trust level TL with which A is granted the permission PT. The risk level can assume the values LOW, MODERATE, HIGH, or EXTREME (see Figure 3.1).

An insider threat can affect the confidentiality, integrity and availability of an organisation's assets. In what follows, we introduce the patterns for insider threats to confidentiality, integrity and availability of an asset.

4.3.1 Patterns of Insider Threats to Confidentiality

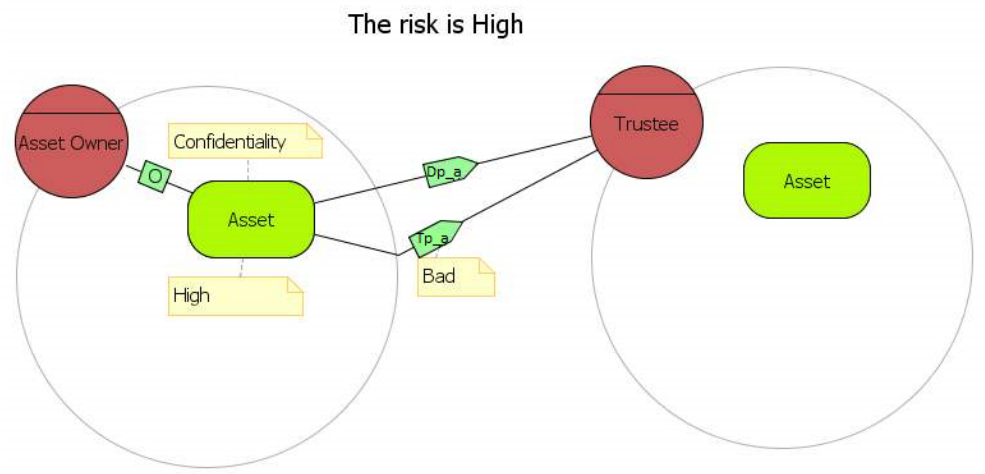

Figure 4.14: Pattern to detect insider threats to confidentiality

Legend: The circles denote the asset owner and the trustee while the ovals contained in the asset owner and trustee compartment denote assets. **Dp_a** and **Tp_a** represent *delegation permission* and *trust of permission* relations where the permission type is *access*. Asset is labelled with the security property that should be satisfied and sensitivity level.

Figure 4.14 illustrates the situation in which the *Trustee* agent is an insider threat for the confidentiality of the *Asset* owned by the *Asset Owner* agent. This is because the *Trustee* has:

- *access* permission on the *Asset* and
- the *Asset Owner* has granted the *access* permission to the *Trustee* agent with medium-low trust level on the *Asset*.

The risk associated with this insider threat is HIGH because the sensitivity level of the *Asset* is HIGH while the trust level with which the *Trustee* has been granted *access* permission is BAD. Note that there are several possible instantiations of this pattern based on the sensitivity level assigned to the *Asset* by the *Asset Owner* and the trust level placed by the *Asset Owner* on the *Trustee* agent with respect to the *access* permission.

4.3.2 Patterns of Insider Threats to Integrity

Figure 4.15 illustrates the situation in which the *Trustee* agent is an insider threat for the integrity of the *Asset* owned by the *Asset Owner* agent. This is because the insider has:

- *modify* permission on the *Asset* and
- the *Asset Owner* has granted the *modify* permission to the *Trustee* agent with medium-low trust level on the *Asset*.

The risk associated with this insider threat is MEDIUM because the sensitivity level of the asset is MEDIUM while the trust level with which the trustee has been granted *modify* permission is NEUTRAL. Note that there are several possible instantiations of this pattern based on the sensitivity level assigned to the *Asset* by the *Asset Owner* and the trust level placed by the *Asset Owner* on the *Trustee* agent with respect to the *modify* permission.

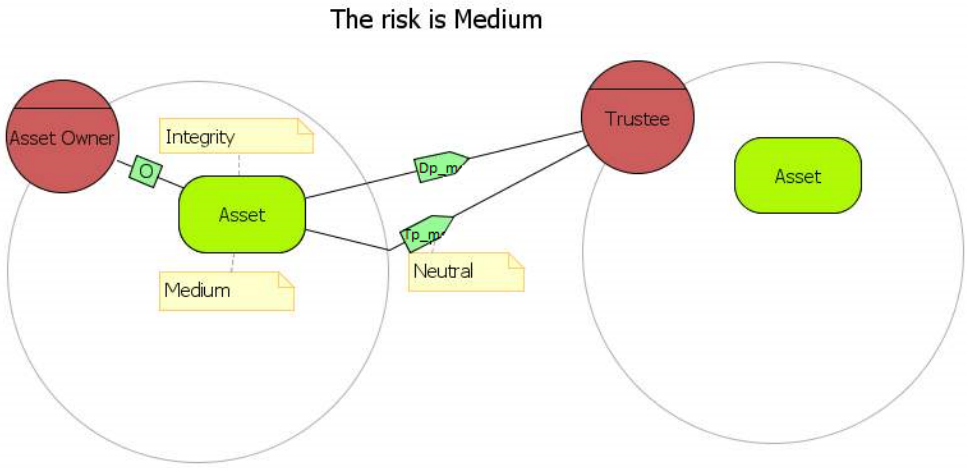

Figure 4.15: Pattern to detect insider threats to integrity

Legend: The circles denote the asset owner and the trustee while the ovals contained in the asset owner and trustee compartment denote assets. **Dp_m** and **Tp_m** represent *delegation permission* and *trust of permission* relations where the permission type is *modify*. Asset is labelled with the security property that should be satisfied and sensitivity level.

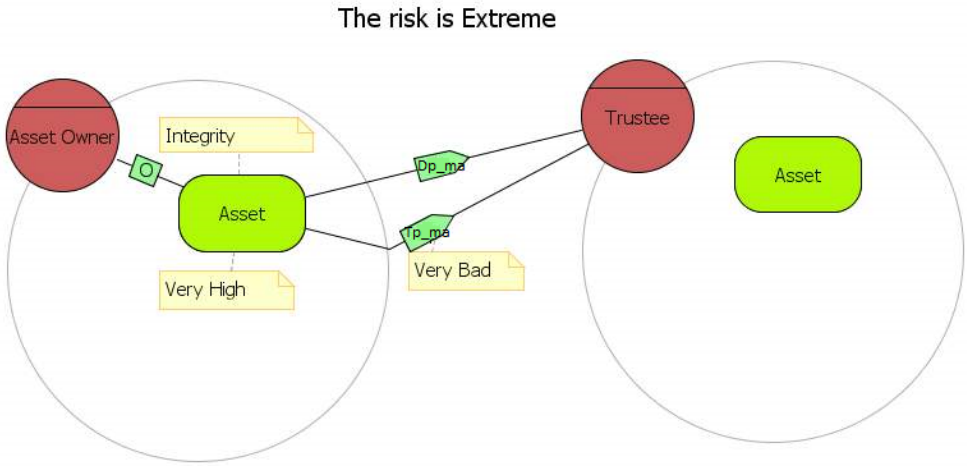

Figure 4.16: Pattern to detect insider threats to availability

Legend: The circles denote the asset owner and the trustee while the ovals contained in the asset owner and trustee compartment denote assets. **Dp_ma** and **Tp_ma** represent *delegation permission* and *trust of permission* relations where the permission type is *manage*. Asset is labelled with the security property that should be satisfied and sensitivity level.

4.3.3 Patterns of Insider Threats to Availability

Figure 4.16 illustrates the situation in which the *Trustee* agent is an insider threat for the availability of the *Asset* owned by the *Asset Owner* agent. This is because the insider has:

- *manage* permission on the *Asset* and
- the *Asset Owner* has granted the *manage* permission to the *Trustee* agent with medium-low trust level on the *Asset*.

The risk associated with this insider threat is EXTREME because the sensitivity of the asset is VERY HIGH and the trust level with which the trustee has been granted *manage* permission is VERY BAD. There are several possible instantiations of this pattern based on the values assumed by the sensitivity level and trust level.

The identification of the insider threats and their risk level is based on a set of axioms in Answer Set Programming (ASP) reported in D6.3 [1]. The modelling and the reasoning based on the above axioms

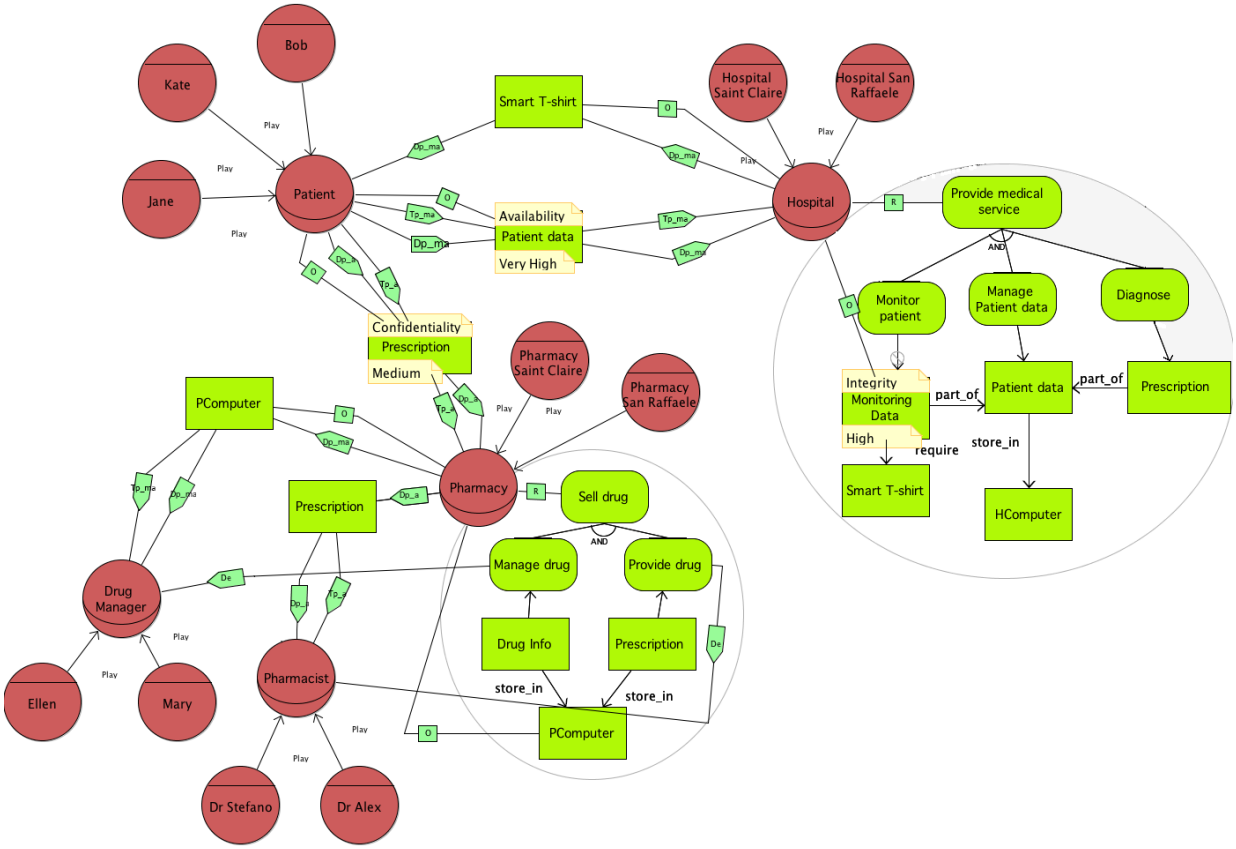

Figure 4.17: Example of Si* model - Patient Monitoring

Legend: The circles denote roles or agents, the ovals denote goals, while the rectangles represent resources. **R** and **O** represent Request and Own. **Dp_a** and **Dp_ma** represent delegation of permission relation where the permission type is *access* and *manage* respectively. Similarly, **Tp_a** and **Tp_ma** represent trust of permission relation where the permission type is *access* and *manage*. Services (e.g., goal, task, resource) that are considered assets are labelled with the security property that should be satisfied and their sensitivity level.

are supported by the Si* tool which is an Eclipse plug-in equipped with a DLV engine. The tool interface allows to draw a Si* model which is automatically translated into ASP specification. The tool also allows to input the rules for insider threat identification so that the problem of identifying insider threats is the same as checking a DLV program that formalize the Si* model and the axioms.

4.3.4 Application to eHealth Case Study

We now illustrate how the patterns can be applied to the eHealth case study. Figure 4.17 shows the Si* model for the Patient Monitoring scenario. The model consists of five roles: the *Hospital*, the *Patient*, the *Pharmacy*, the *Pharmacist*, and the *Drug Manager*.

Patient (*Role*) can be played by three agents *Bob*, *Kate*, and *Jane*. The *Patient* (**O***wns*) the resources *Patient data* and *Prescription*. It delegates to the *Hospital* the *manage* permission on *Patient data*, and it delegates the *access* permission on *Prescription* to the *Pharmacy*. *Pharmacy* (*Role*) can be played by two agents *Pharmacy San Raffaele* and *Pharmacy Saint Claire*. The *Pharmacy* has the intention (**R***equest*) to fulfil the goal *Sell drug* which is (*AND-decomposed*) into subgoals *Manage drug* and *Provide drug*: the fulfilment of *Manage drug* is delegated to the *Drug Manager* while the fulfilment of *Provide drug* is delegated to the *Pharmacist*. The *Pharmacy* (**O***wns*) the resource *PComputer*.

Drug Manager (*Role*) can be played by two agents *Mary* and *Ellen*. To fulfil the goal *Manage drug*, the *Drug Manager* needs to have access to the *PComputer* and thus the *Pharmacy* grants the *manage* permission to the *Drug Manager*. Note that different trustees can be trusted at different trust level for a same permission on an asset. For example, *Mary* is trusted at trust level BAD while *Ellen* is trusted at trust level GOOD for the *manage* permission on the *PComputer*.

Pharmacist (*Role*) can be played by two agents *Dr Stefano* and *Dr Alex*. In order to allow the *Pharmacist* to fulfil the *Provide drug* goal, *Pharmacy* grants the *access* permission on *Prescription* to the *Pharmacist*. *Dr Stefano* and *Dr Alex* are both trusted at trust level GOOD for the *access* permission on *Prescription*.

Hospital (*Role*) can be played by two agents *Hospital San Raffaele* and *Hospital Saint Claire*. The *Hospital* has an intention (**R***equest*) to fulfil the goal *Provide medical service* which is (*AND-decomposed*) into subgoals *Monitor patient*, *Manage patient data*, and *Diagnose*. Some goals can produce or consume resources. For example, the goal *Diagnose* produces the resource *Prescription*. The *Hospital* (**O***wns*) the resource *Smart T-shirt* and delegates to the *Patient* the *manage* permission on it.

Let us assume we want to determine all the possible insider threats for the instance of *Prescription* asset owned by the *Patient Bob*. The reasoning based on the patterns introduced in the previous section reports the following insiders with risk level taken from Figure 3.1 (Section 3.1):

- threat(*Dr Stefano*, asset_instance(service_instance*(Prescription, Bob, Patient*),*Bob,Patient*), confidentiality, MODERATE)
- threat(*Dr Alex*, asset_instance(service_instance(*Prescription, Bob, Patient*),*Bob,Patient*), confidentiality, MODERATE)
- threat(*Mary*, asset_instance(service_instance(*Prescription, Bob, Patient*),*Bob,Patient*), availability, HIGH)
- threat(*Ellen*, asset_instance(service_instance(*Prescription, Bob, Patient*),*Bob,Patient*), availability, MODERATE)

Dr Stefano and *Dr Alex* are two insiders which represent MODERATE risks to the confidentiality of *Prescription* instance owned by *Bob*. *Pharmacy San Raffaele* trusts *Dr Stefano* and *Dr Alex* at the trust level GOOD for the *access* on the *Prescription* instance. For agent who has no direct trust relationship with another agent, we developed rules to compute the trust level between them [35]. We apply those rules to compute the trust values associated with the trust chain from *Bob* to *Dr Stefano* and *Dr Alex* through *Pharmacy San Raffaele*, then both *Dr Stefano* and *Dr Alex* are trusted at trust level GOOD by *Bob* for the *access* permission on *Prescription*. Since the sensitivity of *Bob*'s *Prescription* is MEDIUM, the resulting risk is MODERATE.

Mary is an insider which represents a HIGH risk to the availability of *Prescription* instance owned by *Bob*. *Pharmacy San Raffaele* trusts *Mary* at the trust level BAD for the *manage* permission on the instance of *PComputer* owned by *Pharmacy San Raffaele*. *Bob*'s *Prescription* is stored in *Pharmacy San Raffaele PComputer*. By computing the trust values associated with the trust chain from *Bob* to *Mary* through *Pharmacy San Raffaele*, *Mary* is trusted at trust level BAD by *Bob* for the *manage* permission on *Prescription*. Since the sensitivity of *Bob Prescription* is MEDIUM, the resulting risk is HIGH.

Similarly, *Ellen* is an insider which represents a MODERATE risk to the availability of *Prescription* instance owned by *Bob*.

5 A Pattern-Driven Methodology

In this section we present a systematic methodology to apply the catalogue of patterns presented in the previous section to elicit and analyse security requirements for FI applications. As illustrated in Figure 5.1, the methodology consists of three main phases: *Requirements Elicitation*, *Requirements Analysis* and *Requirements Feedbacks*.

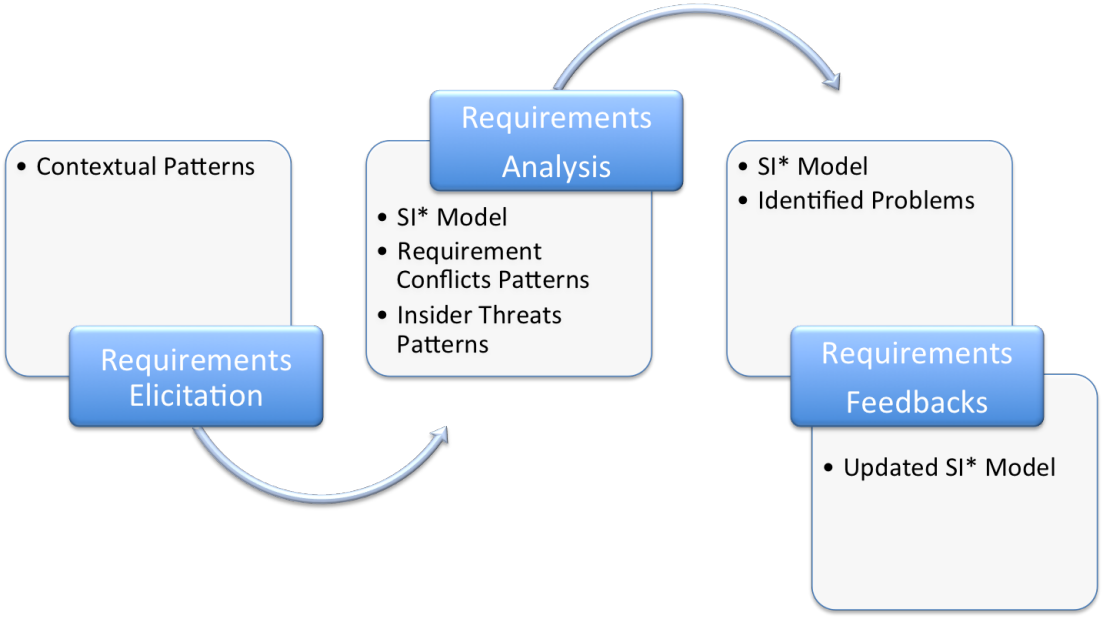

Figure 5.1: Pattern-based methodology for requirements elicitation and analysis

Requirements Elicitation During this phase the security requirements engineer applies trust-aware cloud patterns to identify the main stakeholders of the system-to-be and the trust relationships between them. This information is used to build a Si* model where the stakeholders are represented as actors and trust relationships are mapped to Si* trust of permission or trust of execution relationships.

Pattern-based Requirements Analysis During this phase the security requirements engineer can decide to conduct different kinds of analyses based on the catalogue of patterns introduced in the previous section. The inputs of this phase are thus the catalogue of patterns and the Si* model, while the output is a list of identified problems. The list is piped to next phase. For example, if the security requirements engineer decides to determine if there are goal conflicts present in the Si* models, he will rely on the EMF-IncQuery engine to query the model and check if the goal conflict pattern is present. If this is the case, the engine returns to the security requirements engineer, the actors whose goals are in conflict and the respective goals.

Requirements Feedbacks The security requirements engineer has to resolve the identified problems. The resolution of problems that have been identified with the pattern-based requirements analysis requires to update of the Si* model that may trigger another round of requirements analysis. For example, if the Requirements Analysis phase returns to the security requirements engineer the actors whose goals are in conflict and the colliding goals, the engineer has to modify the requirements model so that the conflict no longer occurs, e.g., delete one of the conflicting goals.

6 From Security Requirements Patterns to Architectural Patterns

Koziolek defines a software system as sustainable “if it can be cost-efficiently maintained and evolved over its entire life-cycle” [24]. Sustainability, often also referred to as ‘evolvability’ or ‘maintainability’, is an important aspect of software development that is becoming of paramount importance for long-living software systems.

To obtain a sustainable software system, the evolutionary aspects must be taken into account from the start of the development process. Change patterns [46] have been recognised as a useful approach to systematically capture and handle evolution of intertwined pairs of artefacts like (security) requirements and software architecture. In particular, change patterns provide a systematic approach that guides the security architect in adapting software architectures in response to changes of trust relationships represented in a requirement model. They capture a particular evolution scenario, and attach architectural solutions to that scenario. As such, a change pattern provides a re-usable connection between a pattern of security requirements change (WP6) and patterns for supporting secure architectural evolution (WP7).

When using change patterns, the security architect has to make two important decisions. First, he has to decide which of the many prospective evolution scenarios (as codified in the catalogues of change patterns) are actually relevant and important to be further considered in a specific context. Second, for each of the considered scenarios, the architect also has to select an appropriate solution out of the candidates that are suggested by the corresponding change pattern.

A current limitation of the change pattern approach is that it does not provide support to the architect in deciding which evolving change scenarios should be considered or discarded and which solutions should be selected for these scenarios. Security architects mainly uses their own experience to prioritise evolution scenarios and to select suitable solutions.

In this chapter, we describe an approach to support the software architect in the prioritization of instances of change patterns and in the selection of suitable architectural solutions. We represent change scenarios in an extended version of the Si* requirements modelling language proposed in [35]. We focus only on scenarios in which the trust level with which an agent is granted a given permission on an asset decreases over time. Then, to prioritise the change scenario instances and to select architectural solutions, we leverage the asset and trust information represented in the Si* requirement model. The trust level and the sensitivity of the asset are used to quantify the risk associated with the actor misusing the granted permission. This risk is then used to determine the relevance and importance of a change scenario instance and to select a suitable solution for the scenario. We illustrate the approach using the eHealth case study from WP11 described earlier (Section 2.1).

6.1 Combining Change Patterns and Risk Information

The basic goal of a change pattern is to provide assistance to software architects to create a sustainable architecture, by presenting suitable solutions for a specific evolution scenario. We propose here an approach that should simplify for the software architect the prioritization of instances of change patterns and the selection of suitable architectural solutions. We only focus here on change patterns for evolving trust-of-permission relationships, and we use Si* to model the change scenarios. To prioritise the change scenarios’ instances and to select architectural solutions, we make use of the asset and trust information represented in the Si* diagrams as presented in Section 3.1. In particular, the trust level with which an agent is granted a given permission on an asset and the sensitivity of the asset are used to quantify the risk associated with the agent misusing the granted permission to cause harm to the asset. The risk is then used to determine the relevance of a change scenario instance and to select a suitable solution for the scenario.

In the rest of this section, we will illustrate the approach with the “Evolving trust-of-permission from external actor” change pattern, which is shown in the top row of Figure 2.5. The situation before change consists of an external party that delegates a permission for some service to the system, and trusts the system for not abusing that permission. The change scenario describes the degradation of this trust relationship over time, i.e., the external party no longer trusts the system with the permission. Therefore,

the trust level that is associated to the trust permission relationship T_P) becomes lower over time.

An architect that wants to use this change pattern has to compare the system's requirement model and the change scenario template to identify possible change scenario instances. In particular, the instances of the change pattern are found by applying pattern matching, where the change scenario template defines the pattern to look for, and the requirements model is the model in which the pattern needs to be found. This process (usually) leads to multiple matches (instances) for a single change scenario. The architect then needs to assess each and every of these matches, to determine which ones are relevant and important enough to spend further effort on. Hence, there is a need for prioritizing the returned matches.

Second, after a relevant instance of a change pattern has been identified, the architect needs to choose between the solutions included with the pattern. This choice, as any architectural decision, needs to take the system context and desired qualities into account. Therefore, the solutions should include the necessary information to make an informed choice.

In the following two subsections, we explain how the risk associated with a trust-of-permission relation in the requirement model can be used by the architect for assessing the importance of change scenario instances and for selecting solutions.

6.1.1 Prioritizing Change Pattern Instances

As mentioned earlier, matching change pattern scenarios with a concrete requirements model can lead to a large amount of instances (matches). However, not all these instances are equally important. To prioritise these concrete change scenarios, each instance needs to be assigned a priority value. For the change patterns that deal with a changing trust-of-permission relationship, a straightforward candidate for the priority is the risk value from Figure 3.1 that is associated with situation after the change has occurred. Change scenarios that lead to a high-risk should be considered more relevant than those leading to a lower risk.

Recall that this risk level is determined by both the trust level and the asset's sensitivity. For the trust level, we use the expected trust level (after evolution), rather than the current trust level, because the purpose of a change pattern is to prepare the system for the occurrence of the change scenario in the future. Hence, the instances with the highest priority will be those involving very sensitive assets and actors that become very poorly trusted with their permissions for accessing these assets, irrespective of their current trust level. These instances should be taken in consideration by the software architect.

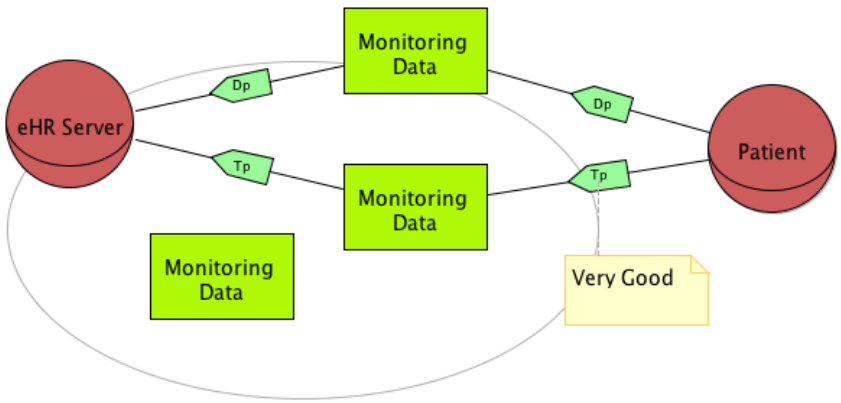

Figure 6.1: Concrete situation before the change happened

Example 6.1 *Figure 6.1 and Figure 6.2 represent a possible instance of our example change pattern in the eHealth case study. In the before situation represented in Figure 6.1, the Patient grants the EHR server permission for accessing the Monitoring Data resource with trust level* VERY GOOD. *A possible evolution scenario is the decrease of this trust level, for example down to* BAD. *In our case study, this would correspond to the scenario where the level of trust from the Patient towards the EHR server decreases over time, for example when the Patient discovers that unauthorised users (a researcher or pharmaceutical*

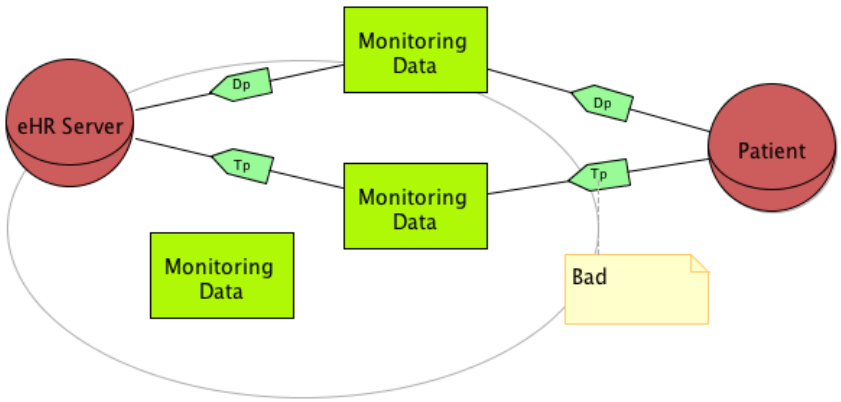

Figure 6.2: Concrete situation after the change happened

company, for instance) have accessed the Patient's Monitoring Data. The EHR server is still entrusted with the permission for accessing the Monitoring Data, but the trust level changes from VERY GOOD *to* BAD*, as shown in Figure 6.2. As we assume that the Monitoring Data has sensitivity level* MEDIUM *and the trust level in the after situation is decreased to* BAD*, the risk level associated with the scenario becomes* HIGH*. Therefore, this scenario will receive a high priority.*

Note that, in general, it may not be possible to directly obtain the priority for a change scenario. This can happen for more complex change scenarios that consider more than one trust relationship at the same time, for example. For such cases, the scenario definition should be augmented with a rule for determining the priority. A simple rule could be to take the maximum risk level associated with all the trust relationships that are part of the scenario. In the current catalogue of change patterns, this is not necessary, as every pattern considers exactly one trust relationship, and the risk level associated with that relationship can be directly used.

As final remark, even when no automated pattern matching techniques are used, the risk levels from the Si* diagrams are still useful with respect to evolution. Indeed, they can be used to pinpoint the most interesting starting points for manual exploration and matching.

6.1.2 Selecting a Suitable Solution

When presented with a relevant evolution scenario, the architect needs to select a solution that will enable that scenario to occur while limiting the impact on the system. The solutions for a single scenario may differ in multiple aspects, including the cost of instantiating them, the (security-wise) guarantees that the solution can offer, as well as the amount of work that is still required when the system needs to be evolved. For the architect, it is therefore not always immediately clear which solution from the catalogue is the best solution in a particular case.

Inspired by the security tactic categories by Bass et al. [6], we therefore propose that each architectural solution of a security-related change pattern is classified as either *resisting abuse*, *detecting abuse*, or *recovering from abuse*. This classification, in a certain sense, captures the "power" of the solutions. Solutions that resist abuse provide the strongest guarantees, as they prevent abuse from happening altogether. Nevertheless, they may not always be applicable or desired, and they can be costly to implement. On the other hand, solutions that only detect abuse mainly serve as a deterrent; by themselves, they do not prevent any misbehaviour. These solutions provide the least guarantees, but may be easier (and cheaper) to implement. In between these two are the solutions that can mitigate abuse. While the abuse may still inflict damage to the system, the implementation of such a solution minimises the impact of this abuse. This includes solutions that allow the system to (partially) recover from the abuse, for example. Sometimes, mitigation solutions require being able to detect abuse in the first place.

As in the previous section, we determine the risk level that is associated with the change scenario based on Figure 3.1. The chosen solution must be strong enough such that it is still appropriate for the new (lower) trust level. Hence, a solution is assessed based on both its classification and the risk

Risk → / ↓ Solution type	Low	Medium	High	Extreme
Solution resists abuse	−	±	+	++
Solution mitigates abuse	+	±	−	−−
Solution detects abuse	++	+	−	−−

Table 6.1: Suitability of a solution given the risk associated with the change (with '++' as 'very suitable', and '−−' as 'not suitable')

Pattern 'Evolving trust-of-permission from external actor'	
Solution 1: Request confirmation	resist
Solution 2: Enable permission monitoring	detect
Pattern 'Evolving trust-of-permission upon external actor'	
Solution 1: Apply least-privilege principle	mitigate
Solution 2: Attribute-based access control	resist
Solution 3: Use permission monitoring	detect
Pattern 'Delegate permission to a service to a trusted actor'	
Solution: Access control	resist
Pattern 'Providing additional service with delegated permission'	
(no architectural solutions)	

Table 6.2: Classification of the solutions for change patterns involving trust-of-permission relationships

level associated with the scenario, where high-risk scenarios demand powerful solutions, while low-risk scenarios lead to less stringent solutions. A proposal for the suitability of a solution with respect to the risk of the scenario can be found in Table 6.1.

Example 6.2 *For the first change pattern in Figure 2.5, two architectural solutions are suggested:*

1. ***Request confirmation*** *The system requests confirmation from the external actor each time it uses its permission for the service. This confirmation can, for instance, be a time-limited token that grants access to the service. This allows the external actor to revoke the permission at any time, or to revoke specific requests based on the information included in them.*

2. ***Enable permission monitoring*** *The usage of the granted permission by the system is monitored by the external actor (or an actor appointed by that external actor) to detect any misbehaviour. By inspecting the obtained information, it can be determined whether the system has abused its permission.*

We classify the first solution (requesting confirmation) as one that resists abuse, and the second solution (enabling permission monitoring) as one that detects abuse.

In our case study, the change scenario instance illustrated in Example 1 leads to a situation with a HIGH *risk level. Applying Table 6.1 to this case, we see that a 'high' risk level indicates that a resistive solution is more preferable. As a consequence, from the two architectural solutions presented above, the first solution is more applicable for this scenario. Hence, the architect may implement this solution by instantiating a service that issues one-time tokens (on behalf of the Patient) whenever the EHR system wants to access that Patient's Monitoring Data. These tokens can be issued based on the Patient's preferences and consent (e.g., the Patient's desire to share Monitoring Data for research purposes).*

We have used only one change pattern as an example in this section, but the solution categories also apply to other change patterns. Table 6.2 shows the assignment of the solution categories to the change patterns from Figure 2.5.

6.2 Discussion

This section provides some noteworthy remarks on the presented approach.

Applicability The approach presented in this chapter relies on the presence of risk-, trust- and sensitivity-related information in a Si* model. As far as we know, Si* is the only notation that has explicitly incorporated this information. While this approach relies on the usage of these enriched Si* models, we believe that it can also be applied using other kinds of requirement models if they incorporate similar information. We believe it is realistic to expect risk-related information to become a part of other types of requirement models as well, because this information identifies the most critical or least trusted parts of the system, and hence conveys important information.

Other architectural constraints The explanation in Sections 6.1.1 and 6.1.2 may appear to suggest that the selection of relevant change pattern instances and adequate solutions is reduced to an algorithmic exercise, and can be completely automated. However, this is not the case. After the architect is presented with the (prioritised) list of change pattern instances, he still needs to select the relevant instances for which the system will provide support. This choice should be based on more factors than just the risk level, for example the likelihood and cost of that particular scenario in the context of the concrete system. It should be noted that the risk information does not reflect these other factors. This means that the architect is not expected to just blindly proceed with the high-risk instances, but he should make a conscious choice based on additional contextual information about the system. Similarly, when selecting a solution, other factors than the risk may favour one solution over the other. Hence, we stress that the proposed approach provides valuable support to the architect, but does not take over (or has the ambition to automate) the core responsibilities of the architect.

Change scenario variants In the conceptual model (Figure 2.4), a change pattern captures a single change scenario. For the patterns presented in this chapter, this single change scenario represents the change of one trust relationship in a particular agent and resource configuration. The presence of the sensitivity and trust levels, however, can also be interpreted as specifying multiple variants of that same scenario. In particular, these variants are differentiated by the initial and final trust levels and the resource sensitivity level, for example one variant where trust decreases from GOOD to BAD for a resource with HIGH sensitivity, and another where trust decreases from VERY GOOD to VERY BAD for a resource with LOW sensitivity. Conceptually, however, all variants represent the same change scenario (a decrease of trust), and are thus bundled within the same change scenario (and the same change pattern). The description of the solutions could still refer to the particular variants of the scenario for which the solution is best suited, following our proposal described in Section 6.1.2.

Benefits for pattern authors The author of a change pattern needs to describe the solutions that are attached to the change scenario of the pattern. It is in the interest of the users of the pattern (the architects) that the set of solutions is complete, i.e., it covers all possible strategies to deal with the occurrence of the change scenario. To accomplish this, mapping the architectural solutions to the categories described in Section 6.1.2 can be a useful technique for a pattern author, as the mapping may trigger additional explorations in order to find missing solutions.

For example, the pattern ‘evolving trust-of-permission from external actor’, which was used as an example in this chapter, currently contains solutions that resist and detect abuse, but no solution that mitigates abuse. A possible solution for mitigation is the use of a trusted third party, which controls the sensitive resource and acts as a mediator between the external party and the system. For example, in the case of credit cards, an online shop can choose to integrate with a service such as PayPal for customers that do not want to provide their credit card details directly to the shop (because they are afraid that the shop’s database may be compromised, for instance). Instead, because the customers do trust PayPal with their credit card information, they can still use the online shop.

7 Relations to other Work Packages

In this section, we give an overview of the relations with other work packages.

WP7 - Secure service architectures and design The cloud pattern is built upon the Cloud Analysis Pattern [11] proposed in WP7. In this deliverable, we use it as a basis for creating a method for calculating trust values in cloud computing scenarios. Moreover, one of our work in the third year by Moyano et al. [31] proposes a UML profile for trust and reputation so that requirements engineer can specify trust and reputation requirements. Using this profile helps to define trust at nodes and components running on these nodes, which provides a valuable input for the framework presented in D7.4 [4]. This framework allows developers to integrate trust and reputation models as part of self-adaptive applications. These applications can then use trust and reputation information to make self-reconfiguration decisions, such as removing a node or changing communication channels among components at runtime.

WP10: Risk and cost aware SDLC. The FI applications have to deal with evolution and changes as one of the important challenges. In this deliverable, we address the system sustainability (which is referred to as evolvability and maintainability) by aiding the architect in the decision making for a suitable solution. We propose the combination of change patterns and risk information to facilitate the solution selection that will enable the changes happen while limiting the impact on the system. Also focusing on the solution selection, the work by Tran [42] proposed an approach to early deal with evolving risks that emerge from the changes of system requirements. This work was presented in D10.4 [3].

WP11 - Future Internet application scenarios The scientific contributions presented in this deliverable have been exemplified on the eHealth case study introduced in D11.3 [2].

8 Conclusions

This deliverable reports the results of WP 6 in the third year of the NESSoS project. We present a framework that supports requirements engineers with modelling and analyses. The framework consists of patterns to elicit stakeholders and their relations, patterns to identify requirements conflicts, and patterns to identify potential threats. We provide extensions to modelling languages (e.g., Si*, UML) to model such patterns.

We propose a meta-model to capture cloud pattern to facilitate the requirements elicitation. We provide several extensions to existing modelling languages, i.e., Si* and UML to capture trust and reputation in requirements. We advocate that, in order to address the security requirements of FI applications, there is the need to make trust information explicit at an early stage in the SDLC. This involves three activities: first, specifying the potential trust relationships among the entities of the application; second, depicting reputation information about these entities, such as which entities can make claims about which other entities; finally, defining the interaction patterns between the business layer and the trust layer.

Based on the extensions on Si* language, we describe several patterns to perform analysis on requirements model based on trust and reputation. In particular, these patterns help to identify potential problems in requirements models such as requirements conflicts, and potential threats in FI applications.

Furthermore, we use the extensions to the Si* language to extend the change patterns approach, which guides software architects with the co-evolution of requirements and architectures. The combination of this trust information and patterns is used to support the software architects, by prioritizing potential evolution scenarios and guiding the selection of a suitable architectural design that can cope with these evolution scenarios. This approach can be employed to address the sustainability for FI applications.

As part of future work, we continue to develop the pattern-driven methodology that applies different patterns presented to elicit and analyse security requirements for FI applications. The preliminary idea of such methodology has been outlined in Section 5.

9 NESSoS WP 6 Third-year Publications

1. K. Beckers, S. Fabbender, M. Heisel, and F. Paci. Combining goal-oriented and problem-oriented requirements engineering methods. In A. Cuzzocrea, C. Kittl, D. Simos, E. Weippl, and L. Xu, editors, *Availability, Reliability, and Security in Information Systems and HCI*, volume 8127 of *Lecture Notes in Computer Science*, pages 178–194. Springer Berlin Heidelberg, 2013.

2. E. Costante, F. Paci, and N. Zannone. Privacy-aware web service composition and ranking. In *Proceedings of the IEEE 20th International Conference on Web Services*, 2013.

3. M. Egea, F. Paci, M. Petrocchi, and N. Zannone. PERSONA: A personalized data protection framework. position paper. In *Proceedings of the 7th IFIP WG 11.11 International Conference on Trust Management*, 2013.

4. F. Moyano, M. C. F. Gago, and J. Lopez. Towards engineering trust-aware future internet systems. In X. Franch and P. Soffer, editors, *Advanced Information Systems Engineering Workshops*, volume 148 of *Lecture Notes in Business Information Processing*, pages 490–501. Springer, 2013.

5. F. Paci, C. Fernandez Gago, and F. Moyano. Detecting insider threats: A trust-aware framework. In *Proceedings of the 8th International Conference on Availability, Reliability and Security*, 2013.

6. L. M. S. Tran. Early dealing with evolving risks in long-life evolving software systems. In X. Franch and P. Soffer, editors, *Advanced Information Systems Engineering Workshops*, volume 148 of *Lecture Notes in Business Information Processing*, pages 518–523. Springer Berlin Heidelberg, 2013.

Bibliography

[1] First Version of the Report describing the Methods and Analysis Techniques along with Prototypical Tools for Methodology Support. NESSoS Deliverable 6.3, October 2012.

[2] Initial version of two case studies, evaluating methodologies. NESSoS Deliverable 11.3, October 2012.

[3] Enhanced Methods for Risk and Cost Aware SDLC. NESSoS Deliverable 10.4, October 2013.

[4] Enhanced set of solutions supporting secure service architectures and design. NESSoS Deliverable 7.4, October 2013.

[5] Y. Asnar, T. Li, F. Massacci, and F. Paci. Computer aided threat identification. In *Proceedings of the 2011 IEEE 13th Conference on Commerce and Enterprise Computing*, pages 145–152. IEEE Computer Society, 2011.

[6] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, second edition, 2003.

[7] K. Beckers, I. Côté, S. Faßbender, M. Heisel, and S. Hofbauer. A pattern-based method for establishing a cloud-specific information security management system. *Requirements Engineering*, pages 1–53, 2013.

[8] K. Beckers, S. Faßbender, and M. Heisel. A meta-model approach to the fundamentals for a pattern language for context elicitation. In *Proceedings of the 18th European Conference on Pattern Languages of Programs (Europlop)*, pages –. ACM, 2013. Accepted for Publication.

[9] K. Beckers, S. Faßbender, M. Heisel, and R. Meis. Pattern-based context establishment for service-oriented architectures. In *Software Service and Application Engineering*, LNCS 7365, pages 81–101. Springer, 2012.

[10] K. Beckers, S. Faßbender, J.-C. Küster, and H. Schmidt. A pattern-based method for identifying and analysing laws in the field of cloud computing compliance. In *Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ)*, volume 7195 of *LNCS*, pages 256–262. Springer, 2012.

[11] K. Beckers, J.-C. Küster, S. Faßbender, and H. Schmidt. Pattern-based support for context establishment and asset identification of the ISO 27000 in the field of cloud computing. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES)*, pages 327–333. IEEE Computer Society, 2011.

[12] G. Bergmann et al. Incremental evaluation of model queries over EMF models. In *Model Driven Engineering Languages and Systems, MODELS'10*. Springer, 2010.

[13] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, SP '96, pages 164–173, Washington, DC, USA, 1996.

[14] S. Chakraborty and I. Ray. TrustBAC: integrating trust relationships into the rbac model for access control in open systems. In *Proceedings of the 11th ACM symposium on Access control models and technologies*, SACMAT '06, pages 49–58, New York, USA, 2006. ACM.

[15] J. Crampton and M. Huth. Towards an access-control framework for countering insider threats. In C. W. Probst, J. Hunker, D. Gollmann, and M. Bishop, editors, *Insider Threats in Cyber Security*, volume 49 of *Advances in Information Security*, pages 173–195. Springer US, 2010.

[16] EU. Directive 95/46/EC of the European Parliament and of the Council of 24 October 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data. Technical report, European Community(EU), 1995.

[17] EU. Directive 2002/58/EC of European Parliament and of the Council of 12 July 2002 concerning the processing of personal data and the protection of privacy in the electronic communications sector (Directive on privacy and electronic communications). Technical report, European Community(EU), 2002.

[18] R. Farmer and B. Glass. *Building Web Reputation Systems*. Yahoo! Press, USA, 1st edition, 2010.

[19] M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1996.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[21] T. Grandison. *Trust management for internet applications*. PhD thesis, University of London, July 2002.

[22] M. Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[23] J. Jürjens. UMLsec: Extending UML for Secure Systems Development. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, UML '02, pages 412–425, London, UK, 2002. Springer-Verlag.

[24] H. Koziolek. Sustainability evaluation of software architectures: a systematic review. In *Proceedings of the joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT symposium–ISARCS on Quality of software architectures–QoSA and architecting critical systems–ISARCS*, pages 3–12. ACM, 2011.

[25] T. Lodderstedt, D. A. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, UML '02, pages 426–441, London, UK, 2002. Springer-Verlag.

[26] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials*, 7:72–93, 2005.

[27] M. S. Lund, B. Solhaug, and K. Stølen. *Model-Driven Risk Analysis - The CORAS Approach*. Springer Berlin Heidelberg, 2011.

[28] M. S. Lund, B. Solhaug, and K. Stølen. *Model-Driven Risk Analysis: The CORAS Approach.* Springer, 2011.

[29] H. Mouratidis and P. Giorgini. Secure Tropos: a Security-Oriented Extension of the Tropos Methodology. *International Journal of Software Engineering and Knowledge Engineering*, 17(2):285–309, 2007.

[30] F. Moyano, C. Fernandez-Gago, and J. Lopez. A conceptual framework for trust models. In *9th International Conference on Trust, Privacy & Security in Digital Business (TrustBus 2012)*, September 2012 In Press.

[31] F. Moyano, M. C. F. Gago, and J. Lopez. Towards engineering trust-aware future internet systems. In X. Franch and P. Soffer, editors, *CAiSE Workshops*, volume 148 of *Lecture Notes in Business Information Processing*, pages 490–501. Springer, 2013.

[32] OECD. OECD Guidelines on the Protection of Privacy and Transborder Flows of Personal Data. Technical report, Organisation for Economic Co-operation and Development (OECD), 1980.

[33] D. Olmedilla, O. F. Rana, B. Matthews, and W. Nejdl. Security and trust issues in semantic grids. In *Semantic Grid: The Convergence of Technologies*, number 05271, Dagstuhl, Germany, 2006.

[34] P. N. Otto and A. I. Antón. Addressing legal requirements in requirements engineering. In *RE*, 2007.

[35] F. Paci, C. Fernandez Gago, and F. Moyano. Detecting insider threats: A trust-aware framework. In *Proceedings of the 8th International Conference on Availability, Reliability and Security*, 2013.

[36] M. Pavlidis, H. Mouratidis, and S. Islam. Modelling Security Using Trust Based Concepts. *IJSSE*, 3(2):36–53, 2012.

[37] L. Rasmusson and S. Jansson. Simulated social control for secure internet commerce. In *Proceedings of the 1996 workshop on New security paradigms (NSPW)*, pages 18–25, New York, USA, 1996. ACM.

[38] B. Schneier. Attack Trees: Modeling Security Threats. *Dr. Dobb's Journal*, December 1999.

[39] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. *Security Patterns: Integrating Security and Systems Engineering*. Wiley, 2006.

[40] G. Sindre and A. L. Opdahl. Eliciting security requirements with misuse cases. *Requir. Eng.*, 10(1):34–44, Jan. 2005.

[41] M. D. D. Toledano. Meta-patterns: Design patterns explained. Technical report, 2002.

[42] L. M. S. Tran. Early dealing with evolving risks in long-life evolving software systems. In X. Franch and P. Soffer, editors, *Advanced Information Systems Engineering Workshops*, volume 148 of *Lecture Notes in Business Information Processing*, pages 518–523. Springer Berlin Heidelberg, 2013.

[43] M. G. Uddin and M. Zulkernine. Umltrust: towards developing trust-aware software. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 831–836, New York, USA, 2008. ACM.

[44] A. van Lamsweerde and E. Letier. Handling Obstacles in Goal-Oriented Requirements Engineering. *IEEE Trans. Softw. Eng.*, 26(10):978–1005, Oct. 2000.

[45] K. Yskout. *Connecting Security Requirements and Software Architecture with Patterns*. PhD thesis, KU Leuven, April 2013.

[46] K. Yskout, R. Scandariato, and W. Joosen. Change patterns: Co-evolving requirements and architecture. *Software & Systems Modeling*, pages 1–24, 2012.

[47] N. Zannone. *A Requirements Engineering Methodology for Trust, Security, and Privacy*. PhD thesis, University of Trento, Italy, 2007.